AD699 Assignment 3: Classification Trees & Random Forests

Student: Saurabh Sharma

Setup

Load all the libraries and configure as needed.

In [1]:
```
# Core libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import confusion_matrix, classification_report, accuracy_score

# OS utils
import os

# Settings
import warnings
warnings.filterwarnings('ignore')  # Silence noisy warnings for cleaner output
sns.set_style('whitegrid')  # Consistent plot aesthetics

# Save the plots in this directory
os.makedirs('outputs', exist_ok=True)

# Random seed
SEED = 750
```

Part 1: Classification Tree

Q1: Load the Dataset

In [2]:
```
# Load data with proper encoding; skip the second header row; use first column as i
df = pd.read_csv('data/Colleges.csv', encoding='latin-1', index_col=0, skiprows=[1]

# Basic shape check for sanity
rows, columns = df.shape
print(f"Dataset shape: Rows({rows})XColumns({columns})")
```

```
# Peek at a random sample to verify loading worked
df.sample(5)
```

```
Dataset shape: Rows(776)XColumns(18)
```

Out[2]:

	Private	Apps	Accept	Enroll	Top10perc	Top25perc	F.Undergrad	P.Undergr
Centenary College of Louisiana	Yes	495	434	210	35	55	775	
Radford University	No	5702	4894	1742	15	37	8077	4
Marymount College Tarrytown	Yes	478	327	117	9	34	731	3
Lambuth University	Yes	831	538	224	15	35	840	3
Moravian College	Yes	1232	955	303	23	58	1241	4

Q2: Describe the Dataset & Summary Statistics

In [3]:
```
print("Dataset Information:")
print(f"  - {len(df)} colleges")
print(f"  - {len(df.columns)} variables")

print(f"\nVariables: {list(df.columns)}")
```

```
Dataset Information:
  - 776 colleges
  - 18 variables

Variables: ['Private', 'Apps', 'Accept', 'Enroll', 'Top10perc', 'Top25perc', 'F.Unde
rgrad', 'P.Undergrad', 'Outstate', 'Room.Board', 'Books', 'Personal', 'PhD', 'Termin
al', 'S.F.Ratio', 'perc.alumni', 'Expend', 'Grad.Rate']
```

In [4]:
```
# Summary statistics
df.describe()
```

Out[4]:

	Apps	Accept	Enroll	Top10perc	Top25perc	F.Undergrad	P.U
count	776.000000	776.000000	776.000000	776.000000	776.000000	776.000000	7
mean	3003.367268	2019.818299	780.048969	27.564433	55.801546	3700.957474	8
std	3872.397321	2452.531771	929.773049	17.650981	19.817081	4853.460439	15
min	81.000000	72.000000	35.000000	1.000000	9.000000	139.000000	
25%	776.000000	603.250000	242.000000	15.000000	41.000000	991.000000	
50%	1557.500000	1109.500000	434.000000	23.000000	54.000000	1707.000000	3
75%	3629.500000	2428.000000	902.250000	35.000000	69.000000	4030.250000	9
max	48094.000000	26330.000000	6392.000000	96.000000	100.000000	31643.000000	218

In [5]:

```python
# Check data types
df.dtypes
```

Out[5]:

```
Private         object
Apps             int64
Accept           int64
Enroll           int64
Top10perc        int64
Top25perc        int64
F.Undergrad      int64
P.Undergrad      int64
Outstate         int64
Room.Board       int64
Books            int64
Personal         int64
PhD              int64
Terminal         int64
S.F.Ratio      float64
perc.alumni      int64
Expend           int64
Grad.Rate        int64
dtype: object
```

In [6]:

```python
# Check for missing values
df.isnull().sum()
```

```
Out[6]: Private        0
        Apps           0
        Accept         0
        Enroll         0
        Top10perc      0
        Top25perc      0
        F.Undergrad    0
        P.Undergrad    0
        Outstate       0
        Room.Board     0
        Books          0
        Personal       0
        PhD            0
        Terminal       0
        S.F.Ratio      0
        perc.alumni    0
        Expend         0
        Grad.Rate      0
        dtype: int64
```

From above it can be observed that there are no null columns.

Q3: Create Yield Variable

```
In [7]: # Create yield = (Enroll / Accept) * 100
        df['yield'] = (df['Enroll'] / df['Accept']) * 100
```

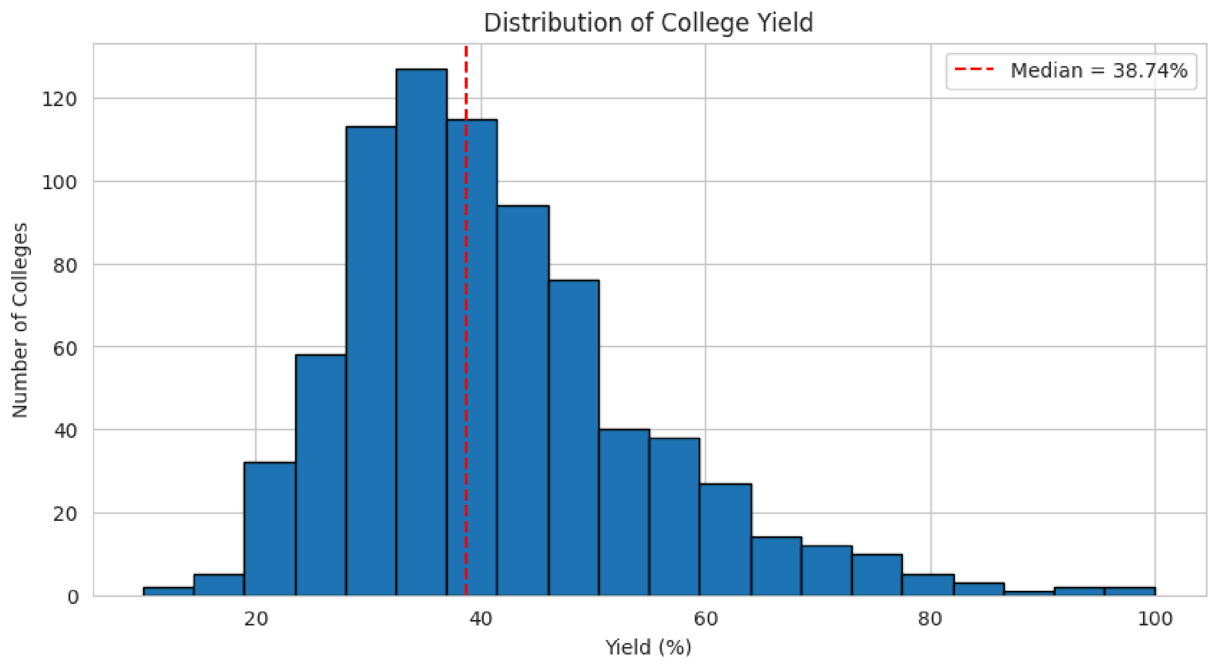
```
In [8]: df['yield'].sample(5, random_state=SEED)
```

```
Out[8]: Point Park College                                 27.822581
        Southwest Baptist University                       58.737420
        Westmont College                                   49.228612
        Grace College and Seminary                         39.018692
        Clinch Valley Coll. of  the Univ. of Virginia      44.563280
        Name: yield, dtype: float64
```

```
In [9]: print("Yield Statistics:")
        print(df['yield'].describe())
```

```
Yield Statistics:
count    776.000000
mean      41.179148
std       13.393036
min        9.975397
25%       31.720105
50%       38.739446
75%       48.561395
max      100.000000
Name: yield, dtype: float64
```

```
In [10]: # Visualize yield distribution
         plt.figure(figsize=(10, 5))
         plt.hist(df['yield'], bins=20, edgecolor='black')
         plt.axvline(df['yield'].median(), color='red', linestyle='--', label=f'Median = {df
         plt.xlabel('Yield (%)')
         plt.ylabel('Number of Colleges')
         plt.title('Distribution of College Yield')
         plt.legend()
         plt.savefig('outputs/yield_distribution.png', dpi=300, bbox_inches='tight')
         plt.show()
```



The yield distribution shows a roughly bell-shaped pattern with a slight right skew, ranging from approximately 10% to 100%. The median yield of 38.74% divides the dataset into equal halves for classification purposes. Most colleges cluster in the 20-50% yield range, with the peak around 30-35%, indicating typical conversion rates in higher education. The right tail extending to 100% represents highly selective institutions where nearly all accepted students enroll, while the left side represents less selective schools where students have multiple options. This distribution suggests yield is influenced by factors like selectivity, reputation, and financial aid, which our classification tree will attempt to model."

In [11]:
```
# Delete the original variables
df = df.drop(['Enroll', 'Accept'], axis=1)

print(f"New shape after dropping Enroll and Accept: {df.shape}")
```

```
New shape after dropping Enroll and Accept: (776, 17)
```

Q4: Convert Yield to Factor (High/Low)

In [12]:
```
# Convert to high/low based on median
median_yield = df['yield'].median()

df['yield_category'] = df['yield'].apply(lambda x: 'high' if x >= median_yield else

print(f"Median yield: {median_yield:.2f}%\n")
print("Class distribution:")
print(df['yield_category'].value_counts())
```

```
Median yield: 38.74%

Class distribution:
yield_category
low     388
high    388
Name: count, dtype: int64
```

In [13]:
```
# Drop the numeric yield column
df = df.drop('yield', axis=1)
```

Q5: Partition Data (60/40 Train/Validation)

In [14]:
```
# Prepare X and y
# X = Features (independent variables): All columns except the target variable
# y = Target (dependent variable): What we're trying to predict
X = df.drop('yield_category', axis=1)  # Remove target column to get features
y = df['yield_category']  # Keep only the target column (high/low)

# Convert Private to numeric (from categorical to binary)
# Machine learning models need numeric input, not text
# 'Yes' becomes 1, 'No' becomes 0
X['Private'] = X['Private'].map({'Yes': 1, 'No': 0})

# Fill any missing values with mean
# Some colleges might have missing data for certain features
# We replace NaN values with the average (mean) of that column
# This prevents errors during model training
X = X.fillna(X.mean())
```

In [15]:
```
# Split the data
X_train, X_val, y_train, y_val = train_test_split(
    X, y, test_size=0.4, random_state=SEED, stratify=y
)

print(f"Training set: {len(X_train)} samples")
print(y_train.value_counts())
print(f"\nValidation set: {len(X_val)} samples")
print(y_val.value_counts())
```

```
Training set: 465 samples
yield_category
low     233
high    232
Name: count, dtype: int64

Validation set: 311 samples
yield_category
high    156
low     155
Name: count, dtype: int64
```

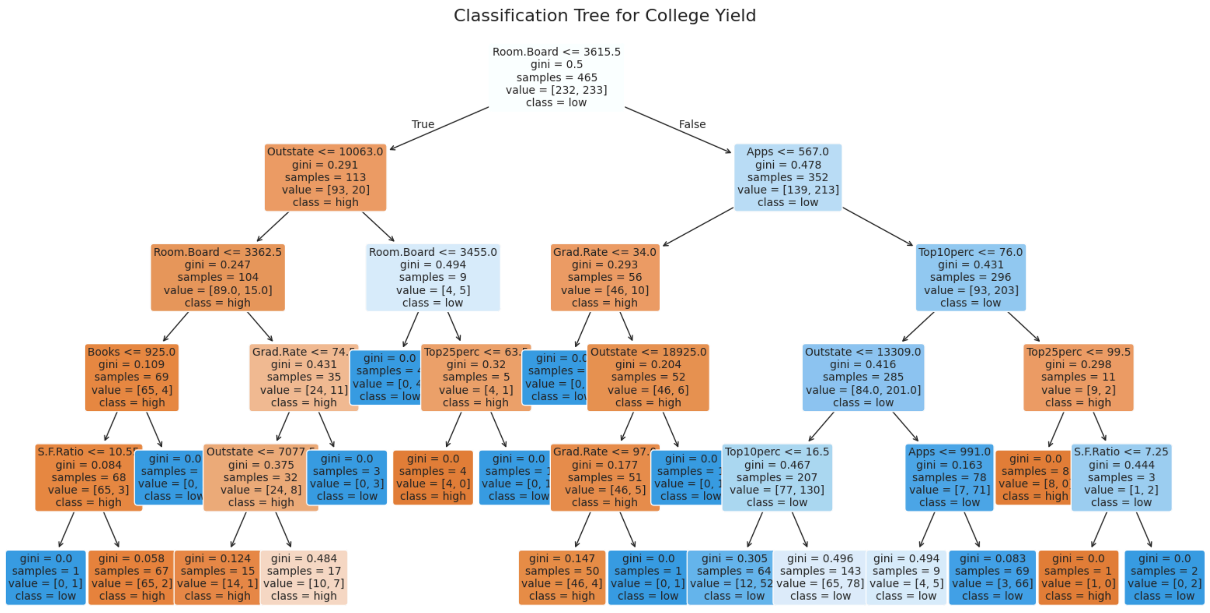
Q6: Build Classification Tree

In [16]:
```python
# Build the tree
dt_model = DecisionTreeClassifier(max_depth=5, random_state=SEED)
dt_model.fit(X_train, y_train)

print(f"Tree depth: {dt_model.get_depth()}")
print(f"Number of leaves: {dt_model.get_n_leaves()}")
```

```
Tree depth: 5
Number of leaves: 20
```

Q7: Display the Tree

In [17]:
```python
plt.figure(figsize=(20, 10))
plot_tree(dt_model,
          feature_names=X.columns,
          class_names=['high', 'low'],
          filled=True,
          rounded=True,
          fontsize=10)
plt.title('Classification Tree for College Yield', fontsize=16)
plt.savefig('outputs/classification_tree.png', dpi=300, bbox_inches='tight')
plt.show()
```


Classification Tree for College Yield
Room.Board <= 3615.5 gini = 0.5 samples = 465 value = [232, 233] class = low
True
False
Outstate <= 10063.0 gini = 0.291 samples = 113 value = [93, 20] class = high
Apps <= 567.0 gini = 0.478 samples = 352 value = [139, 213] class = low
Room.Board <= 3362.5 gini = 0.247 samples = 104 value = [89.0, 15.0] class = high
Room.Board <= 3455.0 gini = 0.494 samples = 9 value = [4, 5] class = low
Grad.Rate <= 34.0 gini = 0.293 samples = 56 value = [46, 10] class = high
Top10perc <= 76.0 gini = 0.431 samples = 296 value = [93, 203] class = low
Books <= 925.0 gini = 0.109 samples = 69 value = [65, 4] class = high
Grad.Rate <= 74. gini = 0.431 samples = 35 value = [24, 11] class = high
gini = 0.0 samples = value = [0, class = low
Top25perc <= 63. gini = 0.32 samples = 5 value = [4, 1] class = high
gini = 0.0 samples = value = [0, class = low
Outstate <= 18925.0 gini = 0.204 samples = 52 value = [46, 6] class = high
Outstate <= 13309.0 gini = 0.416 samples = 285 value = [84.0, 201.0] class = low
Top25perc <= 99.5 gini = 0.298 samples = 11 value = [9, 2] class = high
S.F.Ratio <= 10.5 gini = 0.084 samples = 68 value = [65, 3] class = high
gini = 0.0 samples = value = [0, class = low
Outstate <= 7077 gini = 0.375 samples = 32 value = [24, 8] class = high
gini = 0.0 samples = 3 value = [0, 3] class = low
gini = 0.0 samples = 4 value = [4, 0] class = high
gini = 0.0 samples = value = [0, 1 class = low
Grad.Rate <= 97. gini = 0.177 samples = 51 value = [46, 5] class = high
gini = 0.0 samples = value = [0, class = low
Top10perc <= 16.5 gini = 0.467 samples = 207 value = [77, 130] class = low
Apps <= 991.0 gini = 0.163 samples = 78 value = [7, 71] class = low
gini = 0.0 samples = 8 value = [8, 0 class = high
S.F.Ratio <= 7.25 gini = 0.444 samples = 3 value = [1, 2] class = low
gini = 0.0 samples = 1 value = [0, 1] class = low
gini = 0.058 samples = 67 value = [65, 2] class = high
gini = 0.124 samples = 15 value = [14, 1] class = high
gini = 0.484 samples = 17 value = [10, 7] class = high
gini = 0.147 samples = 50 value = [46, 4] class = high
gini = 0.0 samples = 1 value = [0, 1] class = low
gini = 0.305 samples = 64 value = [12, 52 class = low
gini = 0.496 samples = 143 value = [65, 78] class = low
gini = 0.494 samples = 9 value = [4, 5] class = low
gini = 0.083 samples = 69 value = [3, 66] class = low
gini = 0.0 samples = 1 value = [1, 0] class = high
gini = 0.0 samples = 2 value = [0, 2] class = low

Q8: What Did You See?

In [18]:
```python
# Feature importance
# Create a DataFrame to store and display feature importance scores
# Feature importance tells us which variables had the most influence on the tree's
importance_df = pd.DataFrame({
    'Feature': X.columns,  # Column names from our dataset
    'Importance': dt_model.feature_importances_  # Importance scores from the train
}).sort_values('Importance', ascending=False)  # Sort from most to least important
```

```
print("Feature Importance:")
print(importance_df.head(10))  # Display top 10 most important features
```

```
Feature Importance:
        Feature  Importance
7    Room.Board    0.336933
1          Apps    0.242324
6      Outstate    0.138583
2     Top10perc    0.109527
15    Grad.Rate    0.095497
3     Top25perc    0.032012
12    S.F.Ratio    0.028834
8         Books    0.016289
4   F.Undergrad    0.000000
5   P.Undergrad    0.000000
```

Observations:

The model reveals that `Room.Board` cost is the primary predictor of yield, with lower-cost colleges generally achieving higher yield. The tree's complex structure shows that predicting yield requires considering multiple factors including `tuition`, `applications`, `graduation rates`, and `student quality metrics`, as no single variable perfectly separates `high` from `low` yield colleges.

Q9: Root Node Analysis

In [19]:
```
# Get root node information
# Access the internal tree structure to analyze the first split
tree = dt_model.tree_  # Get the underlying tree structure from the trained model
root_feature_idx = tree.feature[0]  # Index of the feature used at root (position 0
root_threshold = tree.threshold[0]  # The cutoff value used to split at the root
root_feature = X.columns[root_feature_idx]  # Convert index to actual feature name

print("Root Node Split:")
print(f"  Variable: {root_feature}")  # Which feature splits the data first
print(f"  Threshold: {root_threshold:.2f}")  # At what value the split occurs
print(f"  Rule: If {root_feature} <= {root_threshold:.2f} go left, else go right")
```

```
Root Node Split:
  Variable: Room.Board
  Threshold: 3615.50
  Rule: If Room.Board <= 3615.50 go left, else go right
```

Why is the root node significant?

The root node is significant because it represents the single most discriminative feature in the dataset. Room.Board at the `$3,615.50` threshold provides the maximum information gain, meaning this split best separates high-yield from low-yield colleges. As the foundation of the tree, this decision determines the initial grouping from which all subsequent splits are made. The selection of Room.Board suggests that housing affordability is a primary factor influencing student enrollment decisions.

Q10: Which Variables Appeared in the Model?

In [20]:
```
# Find which features were used
# Filter the importance dataframe to separate used and unused features
features_used = importance_df[importance_df['Importance'] > 0]['Feature'].tolist()
features_not_used = importance_df[importance_df['Importance'] == 0]['Feature'].toli

# Display summary of feature usage
print(f"Features used: {len(features_used)} out of {len(X.columns)}")
print(f"\nUsed: {features_used}")  # List of features that appear in the tree
print(f"\nNot used: {features_not_used}")  # Features ignored by the algorithm
```

```
Features used: 8 out of 16

Used: ['Room.Board', 'Apps', 'Outstate', 'Top10perc', 'Grad.Rate', 'Top25perc', 'S.
F.Ratio', 'Books']

Not used: ['F.Undergrad', 'P.Undergrad', 'Private', 'Personal', 'Terminal', 'PhD',
'perc.alumni', 'Expend']
```

Why not all variables?

No, only 8 out of 16 features appear in the model. The decision tree algorithm selectively uses features that maximize information gain at each split. Variables like `F.Undergrad`, `Private`, and `Expend` were excluded because:

1. They're redundant with features already used (e.g., Apps captures size),
2. They have low predictive power for yield,
3. The max_depth=5 constraint limits the number of splits, and
4. The most important features (`Room.Board`, `Apps`, `Outstate`) already explain most of the variation in yield.

This feature selection is actually beneficial—it creates a simpler, more interpretable model focused on the variables that truly matter for predicting enrollment decisions.

Q11: Confusion Matrices & Performance

In [21]:
```
# Make predictions
# Use the trained decision tree to predict yield categories for both datasets
y_train_pred = dt_model.predict(X_train)  # Predictions on training data (data the
y_val_pred = dt_model.predict(X_val)  # Predictions on validation data (unseen data

# Calculate accuracies
# Compare predictions to actual values to measure performance
train_acc = accuracy_score(y_train, y_train_pred)  # % of correct predictions on tr
val_acc = accuracy_score(y_val, y_val_pred)  # % of correct predictions on validati

# Display results
print(f"Training Accuracy: {train_acc:.4f}")  # How well model performs on training
print(f"Validation Accuracy: {val_acc:.4f}")  # How well model generalizes to new d
print(f"Overfitting Gap: {(train_acc - val_acc):.4f}")  # Difference indicates over
```

```
Training Accuracy: 0.7892
Validation Accuracy: 0.6913
Overfitting Gap: 0.0979
```

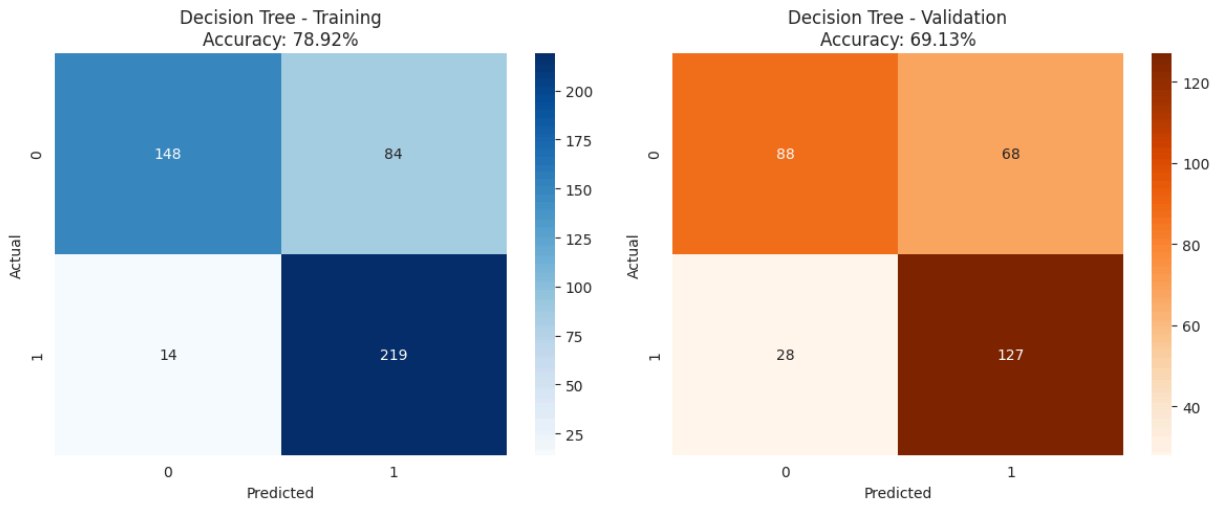
In [22]:
```
# Training confusion matrix
# A confusion matrix shows how many predictions were correct vs incorrect for each
print("Training Set:")
cm_train = confusion_matrix(y_train, y_train_pred)  # Compare actual labels to pred

# Display in a readable DataFrame format with labels
print(pd.DataFrame(cm_train,
                   index=['Actual: high', 'Actual: low'],  # Rows = actual values
                   columns=['Pred: high', 'Pred: low']))  # Columns = predicted val

# Classification report provides precision, recall, F1-score for each class
print(f"\n{classification_report(y_train, y_train_pred)}")
```

```
Training Set:
              Pred: high  Pred: low
Actual: high         148         84
Actual: low           14        219

              precision    recall  f1-score   support

        high       0.91      0.64      0.75       232
         low       0.72      0.94      0.82       233

    accuracy                           0.79       465
   macro avg       0.82      0.79      0.78       465
weighted avg       0.82      0.79      0.78       465
```

In [23]:
```
# Validation confusion matrix
# This is the MOST IMPORTANT evaluation - shows performance on unseen data
print("Validation Set:")
cm_val = confusion_matrix(y_val, y_val_pred)  # Compare actual vs predicted on vali
# Display in DataFrame format for easy reading
print(pd.DataFrame(cm_val,
                   index=['Actual: high', 'Actual: low'],  # Rows = true labels
                   columns=['Pred: high', 'Pred: low']))  # Columns = model predict
# Detailed metrics: precision (accuracy of positive predictions), recall (coverage)
print(f"\n{classification_report(y_val, y_val_pred)}")
```

```
Validation Set:
              Pred: high  Pred: low
Actual: high          88         68
Actual: low           28        127

              precision    recall  f1-score   support

        high       0.76      0.56      0.65       156
         low       0.65      0.82      0.73       155

    accuracy                           0.69       311
   macro avg       0.70      0.69      0.69       311
weighted avg       0.71      0.69      0.69       311
```

In [24]:
```python
# Visualize confusion matrices
# Create side-by-side heatmaps to compare training vs validation performance
fig, axes = plt.subplots(1, 2, figsize=(12, 5))  # 1 row, 2 columns for comparison

# Left plot: Training confusion matrix
sns.heatmap(cm_train, annot=True, fmt='d', cmap='Blues', ax=axes[0])  # Blue colorm
axes[0].set_title(f'Decision Tree - Training\nAccuracy: {train_acc:.2%}')
axes[0].set_ylabel('Actual')  # Y-axis = true labels
axes[0].set_xlabel('Predicted')  # X-axis = predicted labels

# Right plot: Validation confusion matrix (more important!)
sns.heatmap(cm_val, annot=True, fmt='d', cmap='Oranges', ax=axes[1])  # Orange colo
axes[1].set_title(f'Decision Tree - Validation\nAccuracy: {val_acc:.2%}')
axes[1].set_ylabel('Actual')
axes[1].set_xlabel('Predicted')

plt.tight_layout()  # Adjust spacing between plots
plt.savefig('outputs/decision_tree_confusion_matrices.png', dpi=300, bbox_inches='t
plt.show()
```



Performance Assessment:

The decision tree achieved `78.92%` training accuracy and `69.13%` validation accuracy, showing a `9.79%` overfitting gap—acceptable but indicating some memorization of training patterns. The model performs moderately well, significantly better than random guessing (50%), but reveals a critical weakness: it's much better at identifying low-yield colleges (82% recall) than high-yield colleges (56% recall). With 68 false negatives on validation, the model is overly conservative in predicting high yield. While the ~70% validation accuracy is decent, the class imbalance in performance and moderate overfitting suggest this single decision tree has limitations that ensemble methods might address.

Part 2: Random Forest

Q1-2: Same Dataset and Partition

Using the same X_train, X_val, y_train, y_val from above.

Q3: Build Random Forest Model

In [25]:
```
# Build Random Forest
# n_estimators: number of trees in the ensemble
# max_depth: limit tree depth to control overfitting
# random_state: reproducibility
rf_model = RandomForestClassifier(n_estimators=100, max_depth=5, random_state=SEED)
rf_model.fit(X_train, y_train)  # Train on the training split

# Summarize key hyperparameters of the trained model
print(f"Number of trees: {rf_model.n_estimators}")
print(f"Max depth per tree: {rf_model.max_depth}")
```

```
Number of trees: 100
Max depth per tree: 5
```

In [26]:
```
# Feature importance
# Build a DataFrame so we can sort and inspect which features the forest relies on
rf_importance = pd.DataFrame({
    'Feature': X.columns,
    'Importance': rf_model.feature_importances_
}).sort_values('Importance', ascending=False)

# Show the top-ranked features to see drivers of yield predictions
print("Random Forest Feature Importance:")
print(rf_importance.head(10))
```

```
Random Forest Feature Importance:
        Feature  Importance
1          Apps    0.184920
7    Room.Board    0.162566
6      Outstate    0.104279
14       Expend    0.070746
4   F.Undergrad    0.065041
15    Grad.Rate    0.056498
11     Terminal    0.054516
12    S.F.Ratio    0.049775
3     Top25perc    0.043976
2     Top10perc    0.042992
```

Q4: Confusion Matrices & Performance

In [27]:
```
# Make predictions on both seen (train) and unseen (validation) data
y_train_pred_rf = rf_model.predict(X_train)
y_val_pred_rf = rf_model.predict(X_val)

# Calculate accuracy metrics to check fit quality and generalization
train_acc_rf = accuracy_score(y_train, y_train_pred_rf)
val_acc_rf = accuracy_score(y_val, y_val_pred_rf)
```

```
# Report headline metrics and the overfitting gap (train - val)
print(f"Training Accuracy: {train_acc_rf:.4f}")
print(f"Validation Accuracy: {val_acc_rf:.4f}")
print(f"Overfitting Gap: {(train_acc_rf - val_acc_rf):.4f}")
```

```
Training Accuracy: 0.9011
Validation Accuracy: 0.7395
Overfitting Gap: 0.1615
```

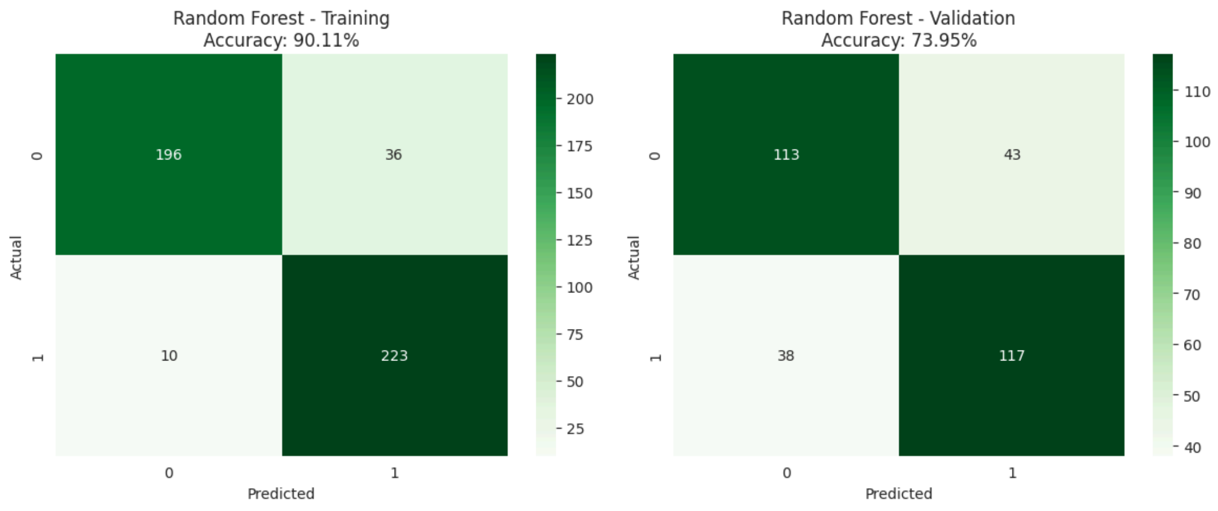
In [28]:
```
# Training confusion matrix helps us see class-wise correctness on data the model s
print("Training Set:")
cm_train_rf = confusion_matrix(y_train, y_train_pred_rf)

# Display counts with readable row/column labels
print(pd.DataFrame(cm_train_rf,
                   index=['Actual: high', 'Actual: low'],
                   columns=['Pred: high', 'Pred: low']))

# Precision/recall/F1 give more detail than accuracy alone
print(f"\n{classification_report(y_train, y_train_pred_rf)}")
```

```
Training Set:
              Pred: high  Pred: low
Actual: high         196         36
Actual: low           10        223

              precision    recall  f1-score   support

        high       0.95      0.84      0.89       232
         low       0.86      0.96      0.91       233

    accuracy                           0.90       465
   macro avg       0.91      0.90      0.90       465
weighted avg       0.91      0.90      0.90       465
```

In [29]:
```
# Validation confusion matrix
print("Validation Set:")
cm_val_rf = confusion_matrix(y_val, y_val_pred_rf)
print(pd.DataFrame(cm_val_rf,
                   index=['Actual: high', 'Actual: low'],
                   columns=['Pred: high', 'Pred: low']))
print(f"\n{classification_report(y_val, y_val_pred_rf)}")
```

```
Validation Set:
              Pred: high  Pred: low
Actual: high         113         43
Actual: low           38        117

              precision    recall  f1-score   support

        high       0.75      0.72      0.74       156
         low       0.73      0.75      0.74       155

    accuracy                           0.74       311
   macro avg       0.74      0.74      0.74       311
weighted avg       0.74      0.74      0.74       311
```

In [30]:
```python
# Visualize confusion matrices for train vs validation side-by-side to spot over/un
fig, axes = plt.subplots(1, 2, figsize=(12, 5))

# Training heatmap: should be high on the diagonal if the model fits training data
sns.heatmap(cm_train_rf, annot=True, fmt='d', cmap='Greens', ax=axes[0])
axes[0].set_title(f'Random Forest - Training\nAccuracy: {train_acc_rf:.2%}')
axes[0].set_ylabel('Actual')
axes[0].set_xlabel('Predicted')

# Validation heatmap: key view for generalization; compare diagonals vs off-diagona
sns.heatmap(cm_val_rf, annot=True, fmt='d', cmap='Greens', ax=axes[1])
axes[1].set_title(f'Random Forest - Validation\nAccuracy: {val_acc_rf:.2%}')
axes[1].set_ylabel('Actual')
axes[1].set_xlabel('Predicted')

plt.tight_layout()
plt.savefig('outputs/random_forest_confusion_matrices.png', dpi=300, bbox_inches='t
plt.show()
```



Performance Assessment:

Random Forest achieved `90.11%` training accuracy and `73.95%` validation accuracy, outperforming the decision tree by `4.82` percentage points on validation data. While the model shows a larger overfitting gap (`16.15%` vs `9.79%`), the superior validation performance is what matters for real-world predictions. Critically, Random Forest dramatically improves high-yield detection—raising recall from `56%` to `72%` and achieves balanced performance across both classes (72-76% recall for both). The model reduces false negatives from 68 to 43, a 37% improvement in the most costly error type. Overall, Random Forest provides meaningfully better predictions with more balanced class performance, justifying its recommendation despite the slightly larger training-validation gap.

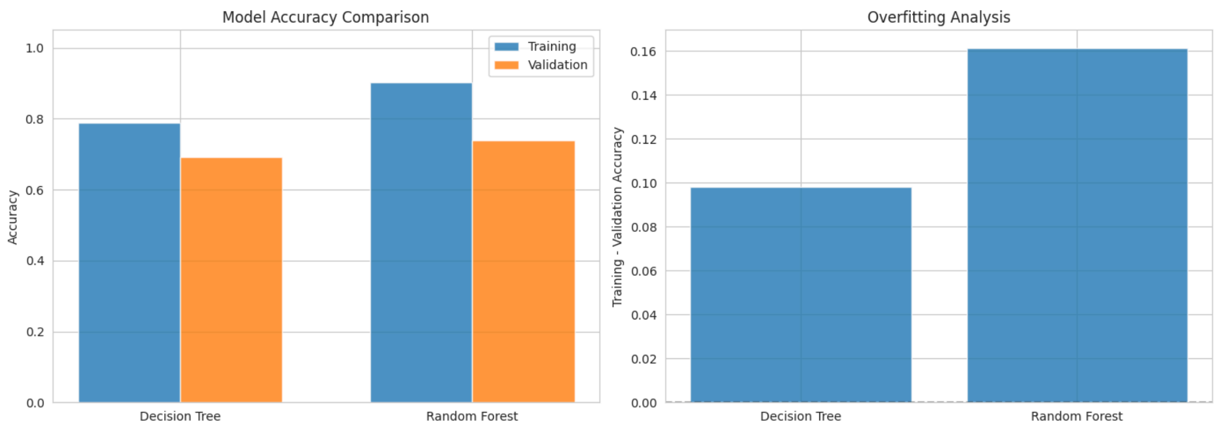
Part 3: Model Comparison

Q1: Compare the Models

In [31]:
```
# Comparison table summarizing key metrics for both models
comparison = pd.DataFrame({
    'Model': ['Decision Tree', 'Random Forest'],
    'Training Acc': [train_acc, train_acc_rf],
    'Validation Acc': [val_acc, val_acc_rf],
    'Overfitting Gap': [train_acc - val_acc, train_acc_rf - val_acc_rf]
})

# Display the table without row indices for readability
print("Model Comparison:")
print(comparison.to_string(index=False))
```

```
Model Comparison:
        Model  Training Acc  Validation Acc  Overfitting Gap
Decision Tree      0.789247        0.691318         0.097929
Random Forest      0.901075        0.739550         0.161525
```

In [32]:
```
# Visualization comparing accuracy and overfitting for both models
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Accuracy comparison: bars for training vs validation to check generalization
x = np.arange(len(comparison))
width = 0.35
axes[0].bar(x - width/2, comparison['Training Acc'], width, label='Training', alpha
axes[0].bar(x + width/2, comparison['Validation Acc'], width, label='Validation', a
axes[0].set_ylabel('Accuracy')
axes[0].set_title('Model Accuracy Comparison')
axes[0].set_xticks(x)
axes[0].set_xticklabels(comparison['Model'])
axes[0].legend()
axes[0].set_ylim([0, 1.05])

# Overfitting comparison: visualize the train-val gap for each model
axes[1].bar(comparison['Model'], comparison['Overfitting Gap'], alpha=0.8)
axes[1].set_ylabel('Training - Validation Accuracy')
axes[1].set_title('Overfitting Analysis')
axes[1].axhline(0, color='black', linestyle='--', alpha=0.3)

plt.tight_layout()
plt.savefig('outputs/model_comparison.png', dpi=300, bbox_inches='tight')  # Save p
plt.show()
```



Q2: Model Recommendation

In [33]:
```
print("RECOMMENDATION: Random Forest\n")

print("Reasons:")
print(f"1. Higher validation accuracy: {val_acc_rf:.2%} vs {val_acc:.2%} (+{(val_ac
print(f"2. More balanced class performance: 72-76% recall vs 56-82% recall")
print(f"3. Reduced false negatives: 43 vs 68 (37% improvement)")
print(f"4. Ensemble robustness through 100 trees averaging")
print(f"\nNote: Larger overfitting gap ({(train_acc_rf - val_acc_rf)*100:.1f}% vs {
print(f"reflects better training fit, not worse generalization - validation accurac
```

```
RECOMMENDATION: Random Forest

Reasons:
1. Higher validation accuracy: 73.95% vs 69.13% (+4.8 points)
2. More balanced class performance: 72-76% recall vs 56-82% recall
3. Reduced false negatives: 43 vs 68 (37% improvement)
4. Ensemble robustness through 100 trees averaging

Note: Larger overfitting gap (16.2% vs 9.8%)
reflects better training fit, not worse generalization - validation accuracy is what
matters!
```

Recommendation

Random Forest is the superior model, achieving 73.95% validation accuracy versus Decision Tree's 69.13% a meaningful 4.82-point improvement that translates to 15 fewer errors.

Addressing the Overfitting Gap:

While Random Forest shows a larger overfitting gap (16.2% vs 9.8%), this does NOT indicate worse generalization. The larger gap occurs because Random Forest learns the training data much better (90% vs 79%), not because it performs worse on validation. The critical metric is validation accuracy, where Random Forest clearly wins (74% vs 69%). The gap exists because:

- Random Forest: Trains to 90%, validates at 74% (16% gap, but 74% is strong)
- Decision Tree: Trains to 79%, validates at 69% (10% gap, but 69% is weak)

Random Forest's superior validation performance makes it the better choice despite the larger numerical gap.