In [1]:

```python
# --- Environment Setup & Imports ---
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from astropy.table import Table
from astropy import units as u
from astropy import coordinates as coord
from astropy.cosmology import Planck18
from scipy.spatial import cKDTree
from sklearn.experimental import enable_halving_search_cv # Enable Halving search
from sklearn.model_selection import train_test_split, StratifiedKFold, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import (
    confusion_matrix, classification_report, accuracy_score,
    precision_score, recall_score, f1_score,
    ConfusionMatrixDisplay, RocCurveDisplay, PrecisionRecallDisplay, make_scorer
)
from sklearn.ensemble import RandomForestClassifier, HistGradientBoostingClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.kernel_approximation import Nystroem
from sklearn.svm import SVC, LinearSVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.pipeline import Pipeline
from sklearn.dummy import DummyClassifier
from sklearn.inspection import permutation_importance
import joblib
import warnings
import time # To time the process
from math import prod # To calculate grid size

# Plotting style configuration
plt.style.use('seaborn-v0_8-talk')
plt.rcParams.update({
    'font.size': 14,
    'axes.labelsize': 16,
    'xtick.labelsize': 12,
    'ytick.labelsize': 12,
    'legend.fontsize': 12,
    'figure.titlesize': 18
})

# Ignore warnings for cleaner output
warnings.filterwarnings('ignore', category=FutureWarning)
warnings.filterwarnings('ignore', category=UserWarning)

print("Libraries imported successfully.")
# Verify Planck18 cosmology is loaded
print(f"Using cosmology: {Planck18.name}")
```

```
Libraries imported successfully.
Using cosmology: Planck18
```

SDSS DR7 Galaxy Catalog

You will be using the SDSS DR7 survey results for the project. We are using the SDSS DR7 main galaxy sample, specifically a volume-limited version of this galaxy catalog. The catalog contains 120,606 galaxies.

In [2]:
```
# --- 2. Load & Inspect SDSS DR7 Galaxy Catalog ---
print("--- Loading Galaxy Catalog ---")
try:
    # Read the ASCII file into an Astropy Table
    galaxy_catalog_ap = Table.read('SDSS_DR7_catlaog_vollim.dat', format='ascii')

    # Convert to pandas DataFrame
    galaxies_df = galaxy_catalog_ap.to_pandas()

    # Print basic info
    print("Galaxy Catalog Info:")
    print(f"Shape: {galaxies_df.shape}")
    print("\nFirst 5 rows:")
    print(galaxies_df.head())
    print("\nSummary Statistics:")
    print(galaxies_df.describe())

except FileNotFoundError:
    print("ERROR: 'SDSS_DR7_catlaog_vollim.dat' not found. Please ensure the file i
except Exception as e:
    print(f"An error occurred: {e}")
```

```
--- Loading Galaxy Catalog ---
Galaxy Catalog Info:
Shape: (120606, 5)

First 5 rows:
           ra       dec  redshift    Rgal  rabsmag
0  171.592148 -1.054439  0.077352  228.33  -20.697
1  174.536224 -1.051174  0.077710  229.37  -20.283
2  239.382782 -0.467646  0.084655  249.51  -20.718
3  239.679092 -0.448756  0.051608  153.14  -20.687
4  239.698471 -0.450346  0.051549  152.97  -20.197

Summary Statistics:
                  ra            dec       redshift           Rgal  \
count  120606.000000  120606.000000  120606.000000  120606.000000
mean      186.511186      25.595987       0.079558     234.479486
std        38.307971      17.690338       0.020233      58.922947
min       109.998665      -3.740457       0.000100       0.300000
25%       156.324512      10.579444       0.067876     200.750000
50%       186.757538      24.325087       0.082948     244.560000
75%       219.139755      38.607632       0.095995     282.270000
max       260.991974      69.879608       0.107000     313.900000

             rabsmag
count  120606.000000
mean      -20.651245
std         0.438362
min       -30.066000
25%       -20.907000
50%       -20.556000
75%       -20.300000
max       -20.090000
```

Right Ascension (RA) & Declination (Dec)

"RA" and "Dec" are common astronomical terms used to specify the position of celestial objects on the sky.

RA (Right Ascension) is like the longitude of the sky, measuring how far east an object is, while Dec (Declination) is like the latitude, measuring how far north or south an object is. Together, they pinpoint the precise location of celestial objects in the night sky, helping astronomers navigate and study them.

Redshifts

This galaxy catalog also has the estiamted redshifts of each galaxy

Comoving Distance

Comoving distance ($R_{\rm gal}$) is called Rgal in the catalog.

R_{gal} = Comoving distance in units of h^{-1} Mpc

(Note: The Hubble constant $H_0 = 100 \cdot h\,\mathrm{km}\,s^{-1}\,\mathrm{Mpc}^{-1}$)

R-band Absolute (AB) Magnitude

The catlaog also contains the AB magnitude of the R-band. This may be useful if you decide to use magnitude cuts.

In [3]:
```
# --- 3. Load & Inspect Ground-Truth Void Catalog ---
print("\n--- Loading Void Catalog ---")

# --- Confirmed settings based on file inspection ---
void_catalog_path = 'V2_VIDE-nsa_v1_0_1_Planck2018_zobovoids.dat'
comment_char = '#'
actual_ra_col_name = 'ra'
actual_dec_col_name = 'dec'
actual_z_col_name = 'redshift'
actual_radius_col_name = 'radius'

try:
    print(f"Attempting to read ASCII file: {void_catalog_path}")
    # Use astropy.table.Table for ASCII.
    void_catalog_ap = Table.read(void_catalog_path, format='ascii',
                                 comment=comment_char)

    # Convert to pandas DataFrame
    voids_df = void_catalog_ap.to_pandas()
    print(f"Successfully read {len(voids_df)} rows.")

    print("Selecting and renaming required columns...")
    # x y z redshift ra dec radius x1 y1 z1 x2 y2 z2 x3 y3 z3 area edge
    voids_df = voids_df[[actual_ra_col_name, actual_dec_col_name, actual_z_col_name

    # Rename for consistency in the rest of the script
    voids_df.rename(columns={
        actual_ra_col_name: 'void_ra',
        actual_dec_col_name: 'void_dec',
        actual_z_col_name: 'void_z',
        actual_radius_col_name: 'void_radius_raw'
    }, inplace=True)

    # --- Unit Conversion Check ---
    print(f"Processing radius column: '{actual_radius_col_name}'")
    print("Input radius units are Mpc/h. Converting to Mpc.")
    h_factor = Planck18.h
    voids_df['void_radius_mpc'] = voids_df['void_radius_raw'] / h_factor

    # Print basic info
    print("\nVoid Catalog Info (Processed):")
    print(f"Number of voids: {len(voids_df)}")
    print("\nBasic Statistics:")
    # Display stats for the columns used going forward
    print(voids_df[['void_ra', 'void_dec', 'void_z', 'void_radius_mpc']].describe()
```

```python
    print("\nFirst 5 rows:")
    print(voids_df.head())
```

```
--- Loading Void Catalog ---
Attempting to read ASCII file: V2_VIDE-nsa_v1_0_1_Planck2018_zobovoids.dat
Successfully read 531 rows.
Selecting and renaming required columns...
Processing radius column: 'radius'
ASSUMPTION: Assuming input radius units are Mpc/h. Converting to Mpc.

Void Catalog Info (Processed):
Number of voids: 531

Basic Statistics:
          void_ra     void_dec      void_z  void_radius_mpc
count  531.000000  531.000000  531.000000       531.000000
mean   182.360005   26.547775    0.083270        26.892926
std     39.416886   17.403740    0.022058        10.070820
min    113.824777   -1.228658    0.014676        14.787883
25%    146.810858   11.310757    0.070083        19.588240
50%    183.813816   25.326738    0.088526        24.049128
75%    214.716981   40.453067    0.101641        32.088390
max    257.577361   66.679157    0.109941        78.502145

First 5 rows:
      void_ra   void_dec    void_z  void_radius_raw  void_radius_mpc
0  148.835073  58.576377  0.102606        23.904054        35.329670
1  151.297597  22.018112  0.103102        26.164603        38.670711
2  168.297676  62.175321  0.100919        22.857079        33.782263
3  193.538452  18.004317  0.091079        24.167234        35.718643
4  140.140987  17.371752  0.102958        23.049027        34.065957
```

In [4]:

```python
# --- 4. Convert Coordinates -> 3D Cartesian ---
print("\n--- Converting Coordinates to Cartesian ---")

def spherical_to_cartesian(ra_deg, dec_deg, dist_mpc):
    """
    Converts spherical coordinates (RA, Dec, Distance) to 3D Cartesian coordinates.

    Args:
        ra_deg (array-like): Right Ascension in degrees.
        dec_deg (array-like): Declination in degrees.
        dist_mpc (array-like): Comoving distance in Mpc.

    Returns:
        tuple: (x, y, z) coordinates in Mpc.
    """
    ra_rad = np.deg2rad(ra_deg)
    dec_rad = np.deg2rad(dec_deg)

    x = dist_mpc * np.cos(dec_rad) * np.cos(ra_rad)
    y = dist_mpc * np.cos(dec_rad) * np.sin(ra_rad)
    z = dist_mpc * np.sin(dec_rad)

    return x, y, z
```

```python
# Apply to Galaxy Catalog (using 'Rgal' which is assumed to be comoving distance in
# --- Unit Conversion Check ---
# If Rgal is in Mpc/h, convert to Mpc.
h = Planck18.h
galaxies_df['Rgal_Mpc'] = galaxies_df['Rgal'] / h # Convert from Mpc/h to Mpc

galaxies_df['gal_x'], galaxies_df['gal_y'], galaxies_df['gal_z'] = spherical_to_car
    galaxies_df['ra'],
    galaxies_df['dec'],
    galaxies_df['Rgal_Mpc'] # Use the converted distance
)
print("Added Cartesian coordinates (gal_x, gal_y, gal_z) to galaxy DataFrame.")
print(galaxies_df[['ra', 'dec', 'Rgal_Mpc', 'gal_x', 'gal_y', 'gal_z']].head())


# Apply to Void Catalog (calculate distance from redshift)
try:
    if 'voids_df' in locals(): # Check if void loading was successful
        voids_df['void_dist_mpc'] = Planck18.comoving_distance(voids_df['void_z']).

        voids_df['void_x'], voids_df['void_y'], voids_df['void_z'] = spherical_to_c
            voids_df['void_ra'],
            voids_df['void_dec'],
            voids_df['void_dist_mpc']
        )
        print("\nAdded Cartesian coordinates (void_x, void_y, void_z) and comoving
        print(voids_df[['void_ra', 'void_dec', 'void_z', 'void_dist_mpc', 'void_rad
    else:
        print("\nSkipping void coordinate conversion as void DataFrame was not load
except Exception as e:
    print(f"\nAn error occurred during void coordinate conversion: {e}")
```

```
--- Converting Coordinates to Cartesian ---
Added Cartesian coordinates (gal_x, gal_y, gal_z) to galaxy DataFrame.
           ra       dec    Rgal_Mpc       gal_x       gal_y     gal_z
0  171.592148 -1.054439  337.466745 -333.783224   49.335559 -6.210196
1  174.536224 -1.051174  339.003843 -337.406819   32.273299 -6.219167
2  239.382782 -0.467646  368.770322 -187.808490 -317.349118 -3.009856
3  239.679092 -0.448756  226.337570 -114.261355 -195.371175 -1.772719
4  239.698471 -0.450346  226.086314 -114.068476 -195.192844 -1.777025

Added Cartesian coordinates (void_x, void_y, void_z) and comoving distance to void D
ataFrame.
      void_ra   void_dec      void_z  void_dist_mpc  void_radius_mpc  \
0  148.835073  58.576377  378.500584     443.554032        35.329670
1  151.297597  22.018112  167.071918     445.644341        38.670711
2  168.297676  62.175321  385.981661     436.443222        33.782263
3  193.538452  18.004317  122.042106     394.844991        35.718643
4  140.140987  17.371752  132.874997     445.037572        34.065957

       void_x      void_y      void_z
0 -197.877990  119.673676  378.500584
1 -362.377112  198.415427  167.071918
2 -199.483204   41.319423  385.981661
3 -365.076403  -87.906262  122.042106
4 -326.039285  272.215100  132.874997
```

In [5]:

```
# --- 5. Label Galaxies by Void Membership ---
print("\n--- Labeling Galaxies by Void Membership ---")

if 'galaxies_df' in locals() and 'voids_df' in locals():
    try:
        # Extract galaxy and void coordinates
        galaxy_coords = galaxies_df[['gal_x', 'gal_y', 'gal_z']].values
        void_center_coords = voids_df[['void_x', 'void_y', 'void_z']].values
        void_radii = voids_df['void_radius_mpc'].values

        # Build KDTree on void centers
        print("Building KDTree on void centers...")
        void_tree = cKDTree(void_center_coords)

        # Query the tree for nearest void for each galaxy
        print("Querying tree for nearest void for each galaxy...")
        dist_to_nearest_void, nearest_void_idx = void_tree.query(galaxy_coords, k=1

        # Get the radius of the nearest void for each galaxy
        radius_of_nearest_void = void_radii[nearest_void_idx]

        # Label galaxies: inside void if distance <= void radius
        galaxies_df['dist_to_nearest_void'] = dist_to_nearest_void
        galaxies_df['nearest_void_idx'] = nearest_void_idx
        galaxies_df['radius_of_nearest_void'] = radius_of_nearest_void
        galaxies_df['is_void'] = (dist_to_nearest_void <= radius_of_nearest_void)

        # Print results
        print("\nVoid Membership Labeling Complete.")
        void_counts = galaxies_df['is_void'].value_counts()
        void_fraction = galaxies_df['is_void'].value_counts(normalize=True)

        print("\nGalaxy Counts by Void Membership:")
        print(void_counts)
        print("\nFraction of Galaxies by Void Membership:")
        print(void_fraction)

        print("\nSample of labeled galaxies:")
        print(galaxies_df[['gal_x', 'gal_y', 'gal_z', 'dist_to_nearest_void', 'radi

    except Exception as e:
        print(f"An error occurred during void labeling: {e}")
else:
    print("Skipping void labeling because galaxy or void DataFrame is missing.")
```

```
--- Labeling Galaxies by Void Membership ---
Building KDTree on void centers...
Querying tree for nearest void for each galaxy...

Void Membership Labeling Complete.

Galaxy Counts by Void Membership:
is_void
False    74336
True     46270
Name: count, dtype: int64

Fraction of Galaxies by Void Membership:
is_void
False    0.616354
True     0.383646
Name: proportion, dtype: float64

Sample of labeled galaxies:
        gal_x       gal_y     gal_z  dist_to_nearest_void  \
0 -333.783224   49.335559 -6.210196             43.856471
1 -337.406819   32.273299 -6.219167             33.130764
2 -187.808490 -317.349118 -3.009856             22.435661
3 -114.261355 -195.371175 -1.772719             41.608877
4 -114.068476 -195.192844 -1.777025             41.795028

   radius_of_nearest_void  is_void
0               30.823867    False
1               30.823867    False
2               15.926156    False
3               32.650417    False
4               32.650417    False
```

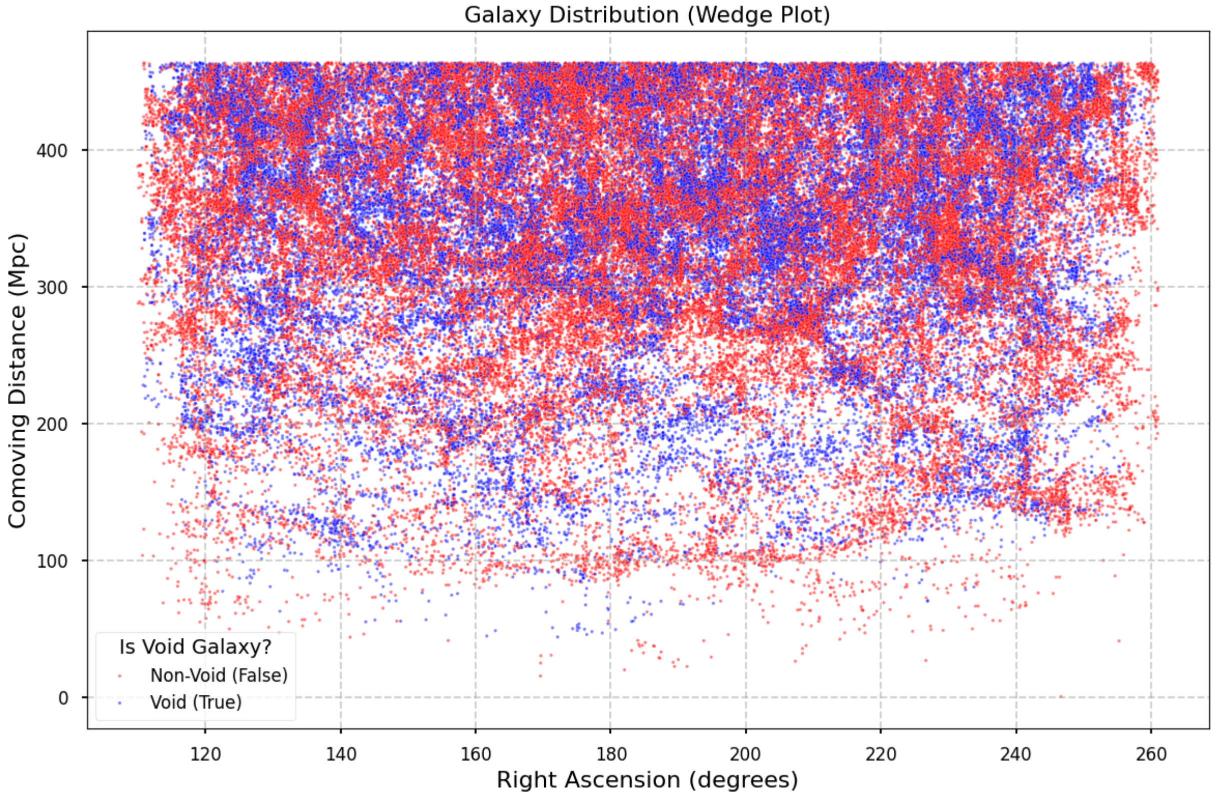
In [6]:

```python
# --- 6. Exploratory Data Analysis (EDA) ---
print("\n--- Performing Exploratory Data Analysis ---")

if 'galaxies_df' in locals() and 'is_void' in galaxies_df.columns:
    # Ensure the column is boolean type before proceeding
    if galaxies_df['is_void'].dtype != 'bool':
        try:
            bool_map = {1: True, 0: False, '1': True, '0': False, True: True, False
            # Apply mapping, keep existing booleans as is, convert others
            galaxies_df['is_void'] = galaxies_df['is_void'].apply(lambda x: bool_ma
            # Final explicit cast to bool, coercing errors to NaT/None might be saf
            galaxies_df['is_void'] = galaxies_df['is_void'].astype(bool)
            print(f"DEBUG: Conversion attempted. New dtype: {galaxies_df['is_void']
        except Exception as convert_err:
            print(f"DEBUG: Failed to convert 'is_void' to bool using map/astype: {c

    try:
        print("DEBUG: Checking 'is_void' column right before plotting:")
        if 'is_void' in galaxies_df.columns:
             print(f"  Data type: {galaxies_df['is_void'].dtype}")
             unique_vals = galaxies_df['is_void'].unique()
             print(f"  Unique values: {unique_vals}")
             # Explicitly check they are boolean True/False
```

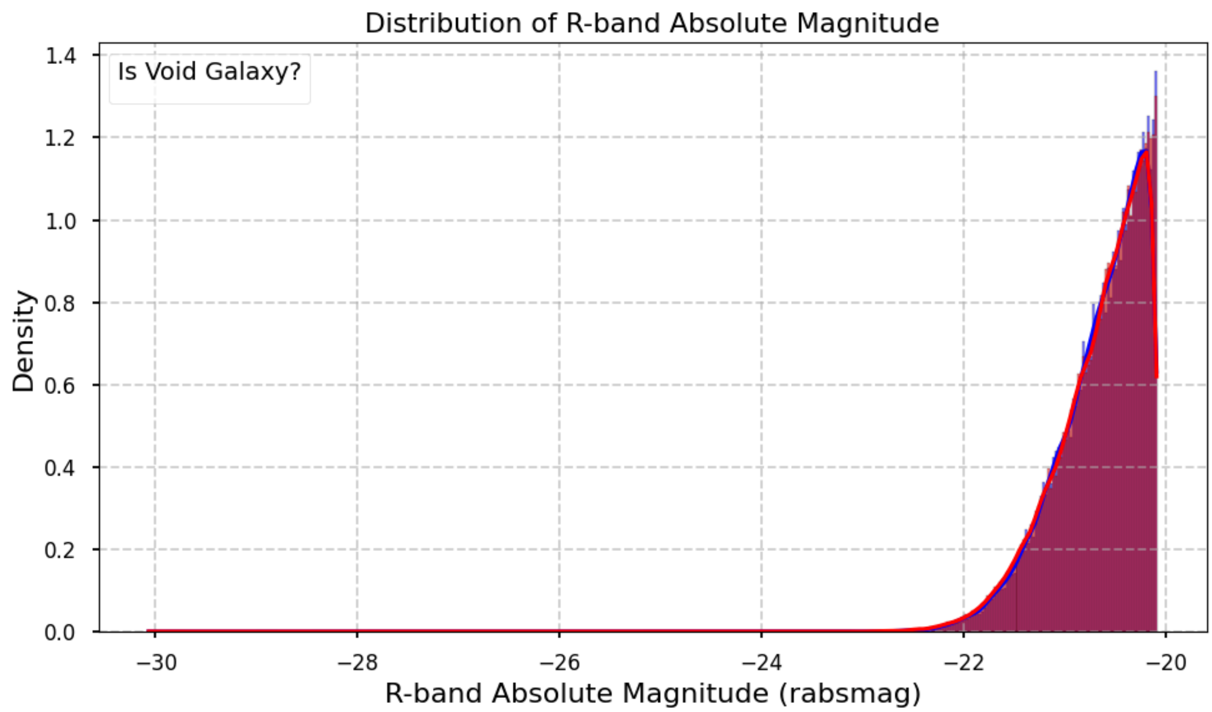
```python
            is_boolean = all(isinstance(v, (bool, np.bool_)) for v in unique_vals)
            print(f"  Are unique values boolean? {is_boolean}")
            if not is_boolean:
                raise TypeError("EDA plotting requires 'is_void' column to be str
        else:
            raise KeyError("'is_void' column not found in DataFrame before plottin

        hue_column = 'is_void' # Use the original boolean column
        plot_palette = {True: 'blue', False: 'red'} # Boolean keys for other plots
        print(f"DEBUG: Using hue column '{hue_column}' with palette keys {list(plot

        # 1. 2D Wedge Plot (RA vs Comoving Distance)
        print("Plotting Wedge Plot...")
        plt.figure(figsize=(12, 8))
        sns.scatterplot(data=galaxies_df, x='ra', y='Rgal_Mpc', hue=hue_column, s=5
        plt.title('Galaxy Distribution (Wedge Plot)')
        plt.xlabel('Right Ascension (degrees)')
        plt.ylabel('Comoving Distance (Mpc)')
        handles, _ = plt.gca().get_legend_handles_labels()
        plt.legend(handles=handles, title='Is Void Galaxy?', labels=['Non-Void (Fal
        plt.grid(True, linestyle='--', alpha=0.6)
        plt.tight_layout()
        plt.show()

        # 2. Histograms of R-band Absolute Magnitude
        print("Plotting Magnitude Histogram...")
        plt.figure(figsize=(10, 6))
        sns.histplot(data=galaxies_df, x='rabsmag', hue=hue_column, kde=True, commo
        plt.title('Distribution of R-band Absolute Magnitude')
        plt.xlabel('R-band Absolute Magnitude (rabsmag)')
        plt.ylabel('Density')
        handles, _ = plt.gca().get_legend_handles_labels()
        plt.legend(handles=handles, title='Is Void Galaxy?', labels=['Non-Void (Fal
        plt.grid(True, linestyle='--', alpha=0.6)
        plt.tight_layout()
        plt.show()

        # 3. Class Balance Bar Chart (Simplified using pandas/matplotlib)
        print("Plotting Class Balance (Simplified)...")
        class_counts = galaxies_df['is_void'].value_counts().sort_index() # Sort Fa
        print("Class Counts:")
        print(class_counts)
        print("\nClass Proportions:")
        print(galaxies_df['is_void'].value_counts(normalize=True).sort_index())

        plt.figure(figsize=(7, 5))
        # Define labels and colors based on sorted index (False, True)
        labels = ['False (Non-Void)', 'True (Void)']
        colors = [plot_palette[False], plot_palette[True]] # Use red for False, blu
        bars = plt.bar(labels, class_counts.values, color=colors)

        # Add counts on top of bars
        for bar in bars:
            yval = bar.get_height()
            plt.text(bar.get_x() + bar.get_width()/2.0, yval, int(yval), va='bottom
```

```
        plt.title('Class Balance: Void vs Non-Void Galaxies')
        plt.xlabel('Is Void Galaxy?')
        plt.ylabel('Number of Galaxies')
        # plt.xticks need not be set again as labels were passed to plt.bar
        plt.grid(True, axis='y', linestyle='--', alpha=0.6)
        plt.tight_layout()
        plt.show()


        # 4. 3D Scatter Plot of a Subsample
        print("Preparing 3D Scatter Plot...")
        subsample_frac = 0.05
        if 'random_state' not in locals(): random_state = 42
        subsample_df = galaxies_df.sample(frac=subsample_frac, random_state=random_

        # Map boolean True/False from the hue_column to colors for the 3D plot
        colors_3d = subsample_df[hue_column].map({True: 'blue', False: 'red'})
        if colors_3d.isnull().any():
            print("Warning: Some values could not be mapped to colors for 3D plot.
            colors_3d = colors_3d.fillna('gray')

        print(f"Plotting 3D Scatter Plot for {len(subsample_df)} ({subsample_frac*1
        fig = plt.figure(figsize=(12, 10))
        ax = fig.add_subplot(111, projection='3d') # Ensure projection='3d' is corr

        ax.scatter(subsample_df['gal_x'], subsample_df['gal_y'], subsample_df['gal_
                   c=colors_3d, s=5, alpha=0.6, marker='.') # Use mapped colors_3d
        ax.set_title(f'3D Distribution of Galaxy Subsample ({subsample_frac*100:.0f
        ax.set_xlabel('X (Mpc)')
        ax.set_ylabel('Y (Mpc)')
        ax.set_zlabel('Z (Mpc)')
        from matplotlib.lines import Line2D
        legend_elements = [Line2D([0], [0], marker='o', color='w', label='Void (Tru
                           Line2D([0], [0], marker='o', color='w', label='Non-Void
        ax.legend(handles=legend_elements, title="Galaxy Type")
        plt.tight_layout()
        plt.show()

    except Exception as e:
        print(f"An error occurred during EDA: {e}")
        import traceback
        traceback.print_exc()
else:
    print("Skipping EDA because galaxy DataFrame ('galaxies_df') or 'is_void' colum
```

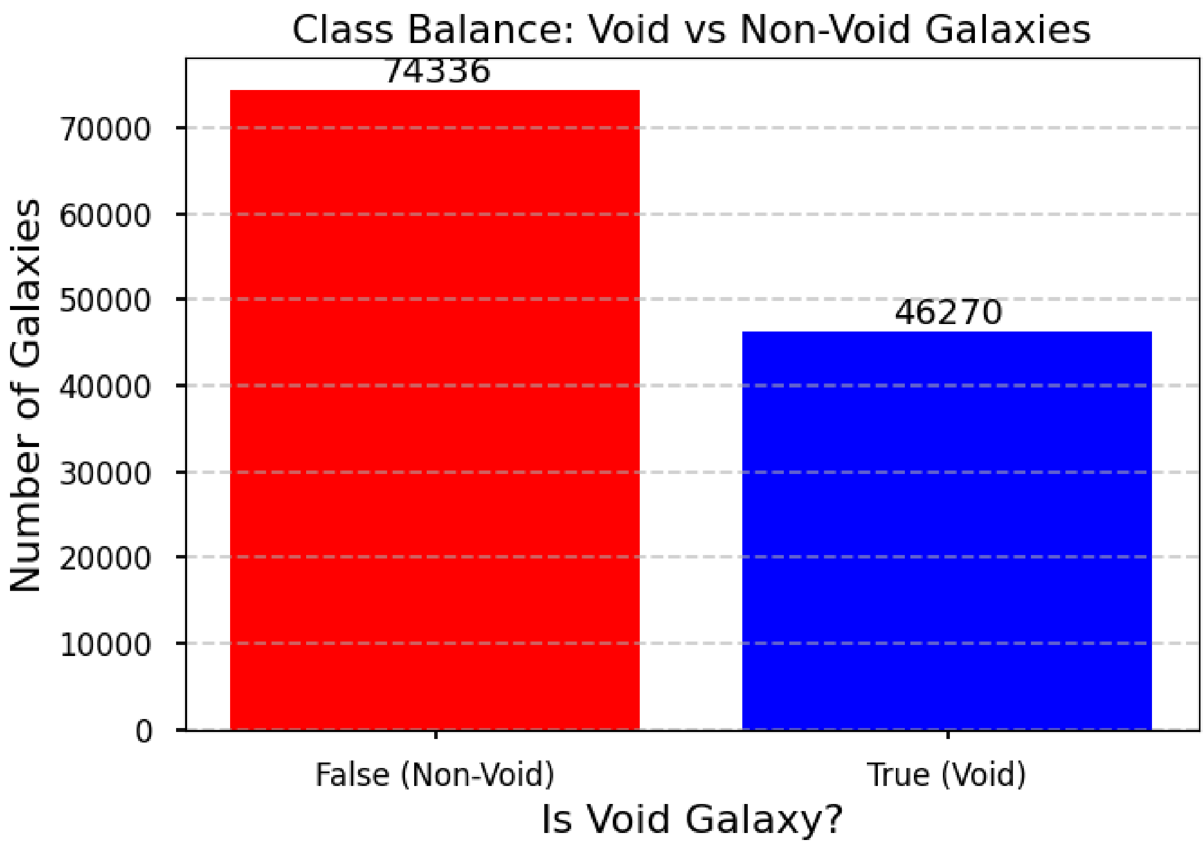
```
--- Performing Exploratory Data Analysis ---
DEBUG: Checking 'is_void' column right before plotting:
  Data type: bool
  Unique values: [False  True]
  Are unique values boolean? True
DEBUG: Using hue column 'is_void' with palette keys [True, False]
Plotting Wedge Plot...
```



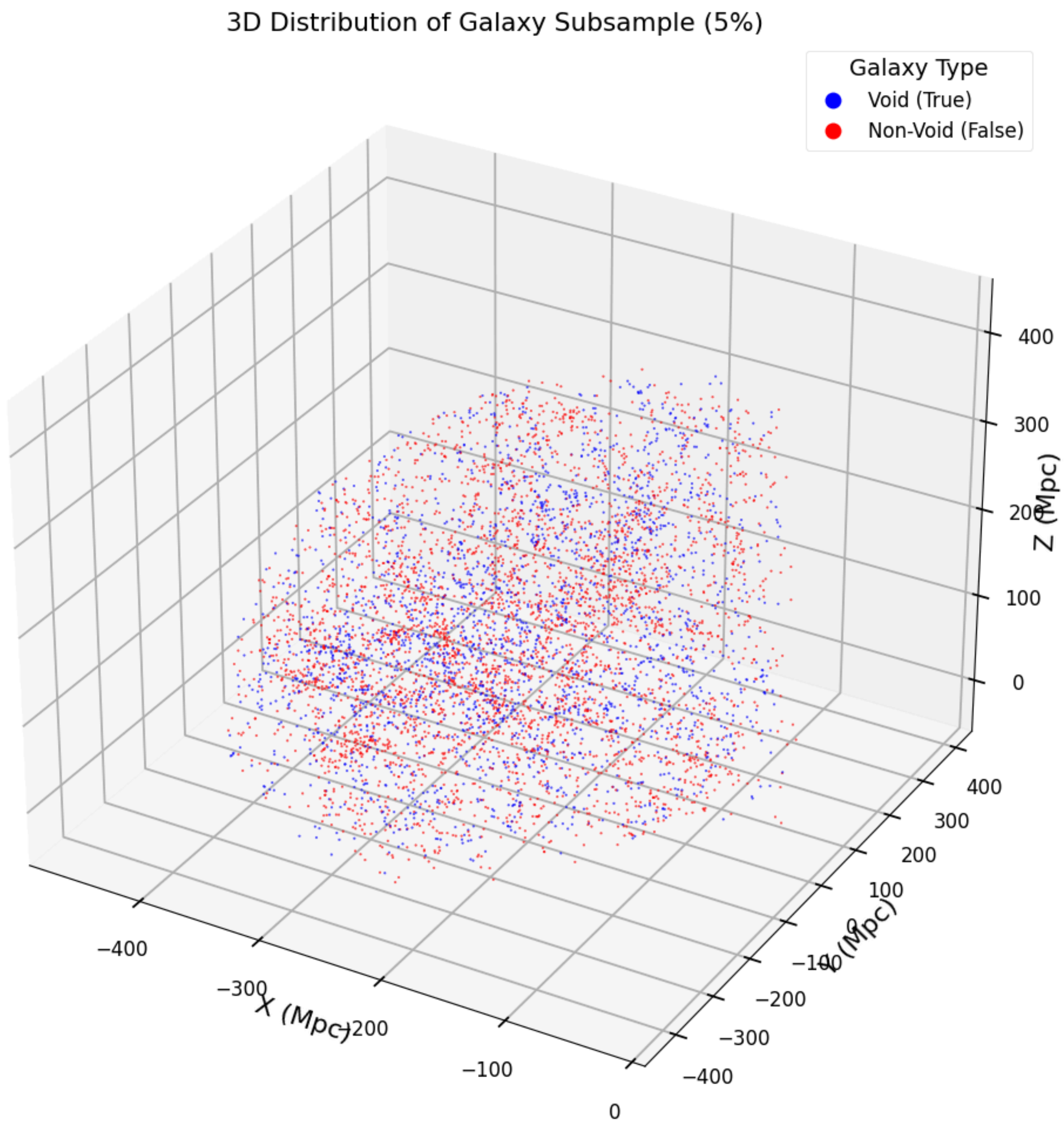

```
Plotting Magnitude Histogram...
```



```
Plotting Class Balance (Simplified)...
Class Counts:
is_void
False    74336
True     46270
Name: count, dtype: int64

Class Proportions:
is_void
False    0.616354
True     0.383646
Name: proportion, dtype: float64
```




```
Preparing 3D Scatter Plot...
Plotting 3D Scatter Plot for 6030 (5.0%) galaxies...
```



In [7]:
```python
# --- 7. Feature Engineering ---
print("\n--- Performing Feature Engineering ---")

if 'galaxies_df' in locals() and 'gal_x' in galaxies_df.columns:
    try:
        # Calculate distance to the Nth nearest neighbor (NN)
        n_neighbors = 5
        print(f"Calculating distance to {n_neighbors}th nearest galaxy neighbor..."

        # Ensure we have the coordinates
        galaxy_coords = galaxies_df[['gal_x', 'gal_y', 'gal_z']].values

        # Build KDTree on galaxy coordinates
        galaxy_tree = cKDTree(galaxy_coords)

        # Query for the N+1 nearest neighbors (index 0 is the point itself)
        # We want the distance to the 5th *other* galaxy, so query for 6 neighbors.
        distances, indices = galaxy_tree.query(galaxy_coords, k=n_neighbors + 1)
```

```python
        # The distance to the Nth neighbor is in the Nth column (index n_neighbors)
        galaxies_df['nn_dist'] = distances[:, n_neighbors]

        print(f"Added '{n_neighbors}th_nn_dist' feature to galaxy DataFrame.")
        print(galaxies_df[['gal_x', 'gal_y', 'gal_z', 'nn_dist']].head())

        # Define feature list for scaling (will be used in the pipeline)
        features_to_scale = ['gal_x', 'gal_y', 'gal_z', 'rabsmag', 'nn_dist']
        print(f"\nFeatures selected for scaling: {features_to_scale}")

    except Exception as e:
        print(f"An error occurred during feature engineering: {e}")
else:
    print("Skipping feature engineering because galaxy DataFrame or coordinate colu
```

```
--- Performing Feature Engineering ---
Calculating distance to 5th nearest galaxy neighbor...
Added '5th_nn_dist' feature to galaxy DataFrame.
        gal_x       gal_y     gal_z    nn_dist
0 -333.783224   49.335559 -6.210196   5.884006
1 -337.406819   32.273299 -6.219167   4.299565
2 -187.808490 -317.349118 -3.009856  13.771429
3 -114.261355 -195.371175 -1.772719   6.512565
4 -114.068476 -195.192844 -1.777025   6.504361

Features selected for scaling: ['gal_x', 'gal_y', 'gal_z', 'rabsmag', 'nn_dist']
```

In [8]:

```python
# --- 8. Train-Test Split & Pipeline Setup ---
print("\n--- Setting up Train-Test Split and Pipeline ---")

if 'galaxies_df' in locals() and 'is_void' in galaxies_df.columns and 'nn_dist' in
    try:
        # Define features (X) and target (y)
        feature_cols = ['gal_x', 'gal_y', 'gal_z', 'rabsmag', 'nn_dist']
        X = galaxies_df[feature_cols].values
        y = galaxies_df['is_void'].astype(int).values # Target variable as integer

        print(f"Feature matrix X shape: {X.shape}")
        print(f"Target vector y shape: {y.shape}")
        print(f"Features used: {feature_cols}")
        print(f"Target variable: is_void (0=Non-Void, 1=Void)")

        # Perform stratified train-test split
        test_size = 0.20
        random_state = 42 # for reproducibility
        X_train, X_test, y_train, y_test = train_test_split(
            X, y,
            test_size=test_size,
            random_state=random_state,
            stratify=y # Important for imbalanced datasets
        )

        print(f"\nSplit data into training ({1-test_size:.0%}) and testing ({test_s
        print(f"X_train shape: {X_train.shape}, y_train shape: {y_train.shape}")
        print(f"X_test shape: {X_test.shape}, y_test shape: {y_test.shape}")
```

```python
        print(f"Void fraction in training set: {np.mean(y_train):.3f}")
        print(f"Void fraction in test set: {np.mean(y_test):.3f}")


        # Build a pipeline with scaling and a placeholder classifier
        # We will replace 'classifier' during hyperparameter tuning
        pipeline = Pipeline([
            ('scaler', StandardScaler()),
            ('classifier', DecisionTreeClassifier(random_state=random_state)) # Pla
        ])

        print("\nPipeline created successfully:")
        print(pipeline)

    except Exception as e:
        print(f"An error occurred during train-test split or pipeline setup: {e}")
else:
    print("Skipping train-test split because required DataFrames/columns are missin
```

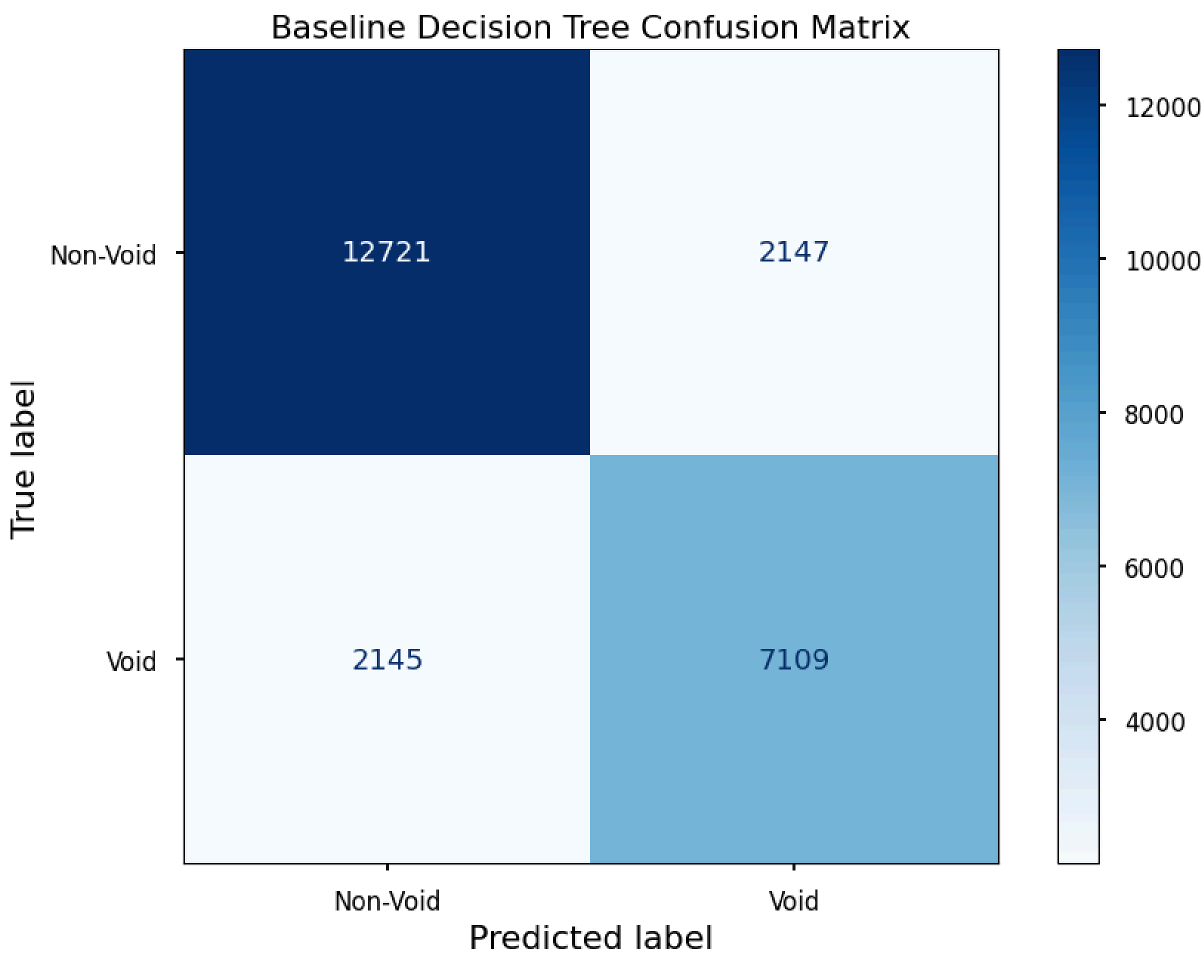
```
--- Setting up Train-Test Split and Pipeline ---
Feature matrix X shape: (120606, 5)
Target vector y shape: (120606,)
Features used: ['gal_x', 'gal_y', 'gal_z', 'rabsmag', 'nn_dist']
Target variable: is_void (0=Non-Void, 1=Void)

Split data into training (80%) and testing (20%) sets.
X_train shape: (96484, 5), y_train shape: (96484,)
X_test shape: (24122, 5), y_test shape: (24122,)
Void fraction in training set: 0.384
Void fraction in test set: 0.384

Pipeline created successfully:
Pipeline(steps=[('scaler', StandardScaler()),
                ('classifier', DecisionTreeClassifier(random_state=42))])
```

In [9]:

```python
# --- 9. Baseline Model & Metrics ---
print("\n--- Evaluating Baseline Model (Default Decision Tree) ---")

if 'pipeline' in locals() and 'X_train' in locals():
    try:
        # Train the baseline pipeline (Scaler + Default Decision Tree)
        pipeline.fit(X_train, y_train)
        print("Baseline pipeline trained.")

        # Evaluate on the test set
        y_pred_baseline = pipeline.predict(X_test)

        print("\nBaseline Model Performance (Decision Tree):")
        print("Confusion Matrix:")
        cm_baseline = confusion_matrix(y_test, y_pred_baseline)
        disp_baseline = ConfusionMatrixDisplay(confusion_matrix=cm_baseline, displa
        disp_baseline.plot(cmap=plt.cm.Blues)
        plt.title('Baseline Decision Tree Confusion Matrix')
        plt.show()

        print("\nClassification Report:")
```

```
        # target_names specify labels for 0 and 1
        print(classification_report(y_test, y_pred_baseline, target_names=['Non-Voi

        # Compare with a trivial baseline (always predict the majority class, likel
        dummy_clf = DummyClassifier(strategy="most_frequent")
        dummy_clf.fit(X_train, y_train)
        y_pred_dummy = dummy_clf.predict(X_test)
        dummy_accuracy = accuracy_score(y_test, y_pred_dummy)

        print(f"\nTrivial Baseline (Always Predict Non-Void) Accuracy: {dummy_accur
        print("Note: Focus on metrics for the 'Void (1)' class (recall, precision,

    except Exception as e:
        print(f"An error occurred during baseline model evaluation: {e}")
else:
    print("Skipping baseline evaluation because pipeline or data splits are missing
```

```
--- Evaluating Baseline Model (Default Decision Tree) ---
Baseline pipeline trained.

Baseline Model Performance (Decision Tree):
Confusion Matrix:
```



```
Classification Report:
               precision    recall  f1-score   support

Non-Void (0)       0.86      0.86      0.86     14868
    Void (1)       0.77      0.77      0.77      9254

    accuracy                           0.82     24122
   macro avg       0.81      0.81      0.81     24122
weighted avg       0.82      0.82      0.82     24122


Trivial Baseline (Always Predict Non-Void) Accuracy: 0.6164
Note: Focus on metrics for the 'Void (1)' class (recall, precision, F1) as it's like
ly the minority class.
```

In [10]:

```
# --- 10. Hyperparameter Tuning with Adaptive & Randomized Search ---
print("\n--- Performing Highly Optimized Hyperparameter Tuning ---")

# Check if necessary variables exist before proceeding
if 'X_train' in locals() and 'y_train' in locals() and 'random_state' in locals():
    try:
        # --- Configuration for Subsampling (ENABLED BY DEFAULT for speed) ---
        USE_SUBSAMPLE_FOR_TUNING = True # Set to True to tune on a smaller subset f
        SUBSAMPLE_SIZE = 30000          # Size of the subset if USE_SUBSAMPLE_FOR_T
        # ------------------------------------------------------------------

        X_tune, y_tune = X_train, y_train # Default to full training set
        if USE_SUBSAMPLE_FOR_TUNING and SUBSAMPLE_SIZE < len(X_train):
            print(f"--- NOTE: Tuning will be performed on a subsample of {SUBSAMPLE
            # Use stratified sampling for the subset
            _, X_tune, _, y_tune = train_test_split(
                X_train, y_train,
                train_size=SUBSAMPLE_SIZE,
                stratify=y_train,
                random_state=random_state
            )
            print(f"Subsample created: X_tune shape {X_tune.shape}, y_tune shape {y
        else:
             print(f"--- NOTE: Tuning will be performed on the full training set ({
             USE_SUBSAMPLE_FOR_TUNING = False


        # --- Define Classifiers including HGB and Nystroem Approx ---
        # TODO Refactor code to delete unecessary or unused models
        print("Defining classifiers...")
        nystroem_svc_pipeline = Pipeline([
            ('nystroem', Nystroem(random_state=random_state)),
            ('linear_svc', LinearSVC(class_weight='balanced', max_iter=8000, random
        ])
        # Include HGB with early stopping enabled by default in its definition
        hgb_base = HistGradientBoostingClassifier(random_state=random_state, early_

        classifiers = {
            'DecisionTree': DecisionTreeClassifier(random_state=random_state),
            'RandomForest': RandomForestClassifier(random_state=random_state, n_job
            'HistGradientBoosting': hgb_base, # Use HGB with early stopping
```

```
        'KNN': KNeighborsClassifier(),
        'LinearSVC': LinearSVC(class_weight='balanced', max_iter=5000, random_s
        'RBFSVC': SVC(kernel='rbf', probability=True, class_weight='balanced',
        'NystroemApproxSVC': nystroem_svc_pipeline
    }
    tree_based_models = {"DecisionTree", "RandomForest", "HistGradientBoosting"

    # --- Define Parameter Grids / Distributions ---
    print("Defining parameter grids/distributions...")
    param_grids = { # Use these as distributions for Randomized Search too
        'DecisionTree': {
            'classifier__max_depth': [10, 20, 30, None],
            'classifier__min_samples_leaf': [5, 10, 20, 50]
        },
        'RandomForest': {
             'classifier__n_estimators': [100, 200, 300],
             'classifier__max_depth': [10, 20, None],
             'classifier__min_samples_leaf': [1, 5, 10],
             'classifier__class_weight': ['balanced', None]
        },
         'HistGradientBoosting': {
             'classifier__max_iter': [100, 200, 300],
             'classifier__max_depth': [5, 10, 15, None],
             'classifier__learning_rate': [0.05, 0.1]
        },
        'KNN': { # Small grid - suitable for HalvingGridSearch
            'classifier__n_neighbors': [5, 11, 21],
            'classifier__weights': ['uniform', 'distance']
        },
        'LinearSVC': { # Small grid
             'classifier__C': [0.01, 0.1, 1, 10]
        },
        'RBFSVC': { # Very small grid
             'classifier__C': [1, 10],
             'classifier__gamma': [0.01, 0.1]
        },
        'NystroemApproxSVC': { # Larger grid - suitable for HalvingRandomSearch
            'classifier__nystroem__n_components': [100, 200, 300],
            'classifier__nystroem__gamma': [0.01, 0.1, 1.0],
            'classifier__linear_svc__C': [0.1, 1, 10]
        }
    }

    # --- Tuning Setup ---
    cv_folds = 3 # Use 3 folds for faster initial tuning pass
    scoring_metric = make_scorer(recall_score, pos_label=1, zero_division=0)
    print(f"Using {cv_folds}-Fold Stratified CV with HalvingGrid/RandomSearchCV
    print(f"Optimizing for: Recall (Void Class = 1)")
    cv = StratifiedKFold(n_splits=cv_folds, shuffle=True, random_state=random_s

    best_estimators = {}
    results_summary = []
    tuning_times = {}
    RANDOM_SEARCH_THRESHOLD = 20
    N_ITER_RANDOM = 25
```

```
    # --- Iterate through classifiers and tune ---
    for name, classifier_obj in classifiers.items():
        start_time = time.time()
        print(f"\n--- Tuning {name} ---")

        # --- Dynamically create pipeline (skip scaler for trees) ---
        if name in tree_based_models:
            steps = [("classifier", classifier_obj)]
            print("Skipping StandardScaler for tree-based model.")
        else:
            steps = [("scaler", StandardScaler()), ("classifier", classifier_ob
        outer_pipeline = Pipeline(steps)
        # ---------------------------------------------------------

        current_param_grid = param_grids[name]
        grid_size = prod(len(v) for v in current_param_grid.values()) if curren
        print(f"Parameter grid size for {name}: {grid_size}")

        # --- Choose HalvingGridSearchCV or HalvingRandomSearchCV ---
        use_randomized = grid_size > RANDOM_SEARCH_THRESHOLD
        search_cv_args = {
            "estimator": outer_pipeline,
            "scoring": scoring_metric,
            "cv": cv,
            "factor": 3,
            "min_resources": "smallest",
            "n_jobs": -1,
            "verbose": 2,
            "random_state": random_state # Ensures reproducibility across runs/
        }

        if use_randomized:
            print(f"Using HalvingRandomSearchCV (grid size {grid_size} > {RANDO
            search_cv = HalvingRandomSearchCV(
                **search_cv_args,
                param_distributions=current_param_grid,
                n_candidates='exhaust', # Explore specified candidates per iter
            )
        else:
            print(f"Using HalvingGridSearchCV (grid size {grid_size} <= {RANDOM
            search_cv = HalvingGridSearchCV(
                **search_cv_args,
                param_grid=current_param_grid,
            )
        # ----------------------------------------------------------

        # Fit the search object on the tuning data (full or subsample)
        print(f"Fitting {type(search_cv).__name__} for {name} using {len(y_tune
        search_cv.fit(X_tune, y_tune)
        end_time = time.time()
        tuning_times[name] = end_time - start_time
        print(f"Finished fitting {name} in {tuning_times[name]:.2f} seconds.")

        # Store results - best estimator found by the search
        best_estimators[name] = search_cv.best_estimator_
        results_summary.append({
```

```
                'Model': name,
                'Best Score (Recall)': search_cv.best_score_,
                'Best Params': search_cv.best_params_,
                'Search Type': type(search_cv).__name__ # Record which search was u
            })

            print(f"Best {name} Recall (CV): {search_cv.best_score_:.4f}")
            print(f"Best {name} Parameters: {search_cv.best_params_}")

            if USE_SUBSAMPLE_FOR_TUNING:
                 print(f"--- Refitting best {name} estimator on FULL training data
                 start_refit_time = time.time()
                 try:
                     # Ensure the best_estimator_ pipeline is used for refitting
                     best_estimators[name].fit(X_train, y_train) # Refit on full da
                     end_refit_time = time.time()
                     print(f"Finished refitting in {end_refit_time - start_refit_ti
                 except Exception as refit_e:
                     print(f"ERROR during refitting best {name}: {refit_e}")


        # --- Display summary of tuning results ---
        print("\n--- Hyperparameter Tuning Summary ---")
        results_df = pd.DataFrame(results_summary).set_index('Model')
        results_df['Tuning Time (s)'] = results_df.index.map(tuning_times)

        # Reorder columns for clarity
        cols_order = ['Best Score (Recall)', 'Best Params', 'Search Type', 'Tuning
        results_df = results_df[cols_order]

        with pd.option_context('display.max_colwidth', None):
            print(results_df)

    except NameError as ne:
        print(f"Skipping hyperparameter tuning because a required variable is missi
        import traceback; traceback.print_exc()
    except ImportError:
        print("ImportError: Make sure scikit-learn is up-to-date (>= 1.0 for Halvin
        print("Try: pip install -U scikit-learn")
        import traceback; traceback.print_exc()
    except Exception as e:
        print(f"An error occurred during hyperparameter tuning: {e}")
        import traceback; traceback.print_exc()
else:
    missing_vars = []
    if 'X_train' not in locals(): missing_vars.append('X_train')
    if 'y_train' not in locals(): missing_vars.append('y_train')
    if 'random_state' not in locals(): missing_vars.append('random_state')
    print(f"Skipping hyperparameter tuning because required variables are missing:
```

```
--- Performing Highly Optimized Hyperparameter Tuning ---
--- NOTE: Tuning will be performed on a subsample of 30000 data points. ---
Subsample created: X_tune shape (66484, 5), y_tune shape (66484,)
Defining classifiers...
Defining parameter grids/distributions...
Using 3-Fold Stratified CV with HalvingGrid/RandomSearchCV.
Optimizing for: Recall (Void Class = 1)

--- Tuning DecisionTree ---
Skipping StandardScaler for tree-based model.
Parameter grid size for DecisionTree: 16
Using HalvingGridSearchCV (grid size 16 <= 20).
Fitting HalvingGridSearchCV for DecisionTree using 66484 samples...
n_iterations: 3
n_required_iterations: 3
n_possible_iterations: 8
min_resources_: 12
max_resources_: 66484
aggressive_elimination: False
factor: 3
----------
iter: 0
n_candidates: 16
n_resources: 12
Fitting 3 folds for each of 16 candidates, totalling 48 fits
----------
iter: 1
n_candidates: 6
n_resources: 36
Fitting 3 folds for each of 6 candidates, totalling 18 fits
----------
iter: 2
n_candidates: 2
n_resources: 108
Fitting 3 folds for each of 2 candidates, totalling 6 fits
Finished fitting DecisionTree in 4.04 seconds.
Best DecisionTree Recall (CV): 0.3974
Best DecisionTree Parameters: {'classifier__max_depth': None, 'classifier__min_sampl
es_leaf': 5}
--- Refitting best DecisionTree estimator on FULL training data (96484 points)... --
-
Finished refitting in 1.04 seconds.

--- Tuning RandomForest ---
Skipping StandardScaler for tree-based model.
Parameter grid size for RandomForest: 54
Using HalvingRandomSearchCV (grid size 54 > 20).
Fitting HalvingRandomSearchCV for RandomForest using 66484 samples...
n_iterations: 4
n_required_iterations: 4
n_possible_iterations: 8
min_resources_: 12
max_resources_: 66484
aggressive_elimination: False
factor: 3
----------
```

```
iter: 0
n_candidates: 54
n_resources: 12
Fitting 3 folds for each of 54 candidates, totalling 162 fits
----------
iter: 1
n_candidates: 18
n_resources: 36
Fitting 3 folds for each of 18 candidates, totalling 54 fits
----------
iter: 2
n_candidates: 6
n_resources: 108
Fitting 3 folds for each of 6 candidates, totalling 18 fits
----------
iter: 3
n_candidates: 2
n_resources: 324
Fitting 3 folds for each of 2 candidates, totalling 6 fits
Finished fitting RandomForest in 15.39 seconds.
Best RandomForest Recall (CV): 0.4057
Best RandomForest Parameters: {'classifier__n_estimators': 100, 'classifier__min_sam
ples_leaf': 5, 'classifier__max_depth': None, 'classifier__class_weight': 'balance
d'}
--- Refitting best RandomForest estimator on FULL training data (96484 points)... --
-
Finished refitting in 4.02 seconds.

--- Tuning HistGradientBoosting ---
Skipping StandardScaler for tree-based model.
Parameter grid size for HistGradientBoosting: 24
Using HalvingRandomSearchCV (grid size 24 > 20).
Fitting HalvingRandomSearchCV for HistGradientBoosting using 66484 samples...
n_iterations: 3
n_required_iterations: 3
n_possible_iterations: 8
min_resources_: 12
max_resources_: 66484
aggressive_elimination: False
factor: 3
----------
iter: 0
n_candidates: 24
n_resources: 12
Fitting 3 folds for each of 24 candidates, totalling 72 fits
----------
iter: 1
n_candidates: 8
n_resources: 36
Fitting 3 folds for each of 8 candidates, totalling 24 fits
----------
iter: 2
n_candidates: 3
n_resources: 108
Fitting 3 folds for each of 3 candidates, totalling 9 fits
Finished fitting HistGradientBoosting in 1.46 seconds.
```

```
Best HistGradientBoosting Recall (CV): 0.4487
Best HistGradientBoosting Parameters: {'classifier__max_iter': 200, 'classifier__max
_depth': None, 'classifier__learning_rate': 0.1}
--- Refitting best HistGradientBoosting estimator on FULL training data (96484 point
s)... ---
Finished refitting in 1.07 seconds.

--- Tuning KNN ---
Parameter grid size for KNN: 6
Using HalvingGridSearchCV (grid size 6 <= 20).
Fitting HalvingGridSearchCV for KNN using 66484 samples...
n_iterations: 2
n_required_iterations: 2
n_possible_iterations: 8
min_resources_: 12
max_resources_: 66484
aggressive_elimination: False
factor: 3
----------
iter: 0
n_candidates: 6
n_resources: 12
Fitting 3 folds for each of 6 candidates, totalling 18 fits
----------
iter: 1
n_candidates: 2
n_resources: 36
Fitting 3 folds for each of 2 candidates, totalling 6 fits
Finished fitting KNN in 0.52 seconds.
Best KNN Recall (CV): 0.2778
Best KNN Parameters: {'classifier__n_neighbors': 5, 'classifier__weights': 'distanc
e'}
--- Refitting best KNN estimator on FULL training data (96484 points)... ---
Finished refitting in 0.10 seconds.

--- Tuning LinearSVC ---
Parameter grid size for LinearSVC: 4
Using HalvingGridSearchCV (grid size 4 <= 20).
Fitting HalvingGridSearchCV for LinearSVC using 66484 samples...
n_iterations: 2
n_required_iterations: 2
n_possible_iterations: 8
min_resources_: 12
max_resources_: 66484
aggressive_elimination: False
factor: 3
----------
iter: 0
n_candidates: 4
n_resources: 12
Fitting 3 folds for each of 4 candidates, totalling 12 fits
----------
iter: 1
n_candidates: 2
n_resources: 36
Fitting 3 folds for each of 2 candidates, totalling 6 fits
```

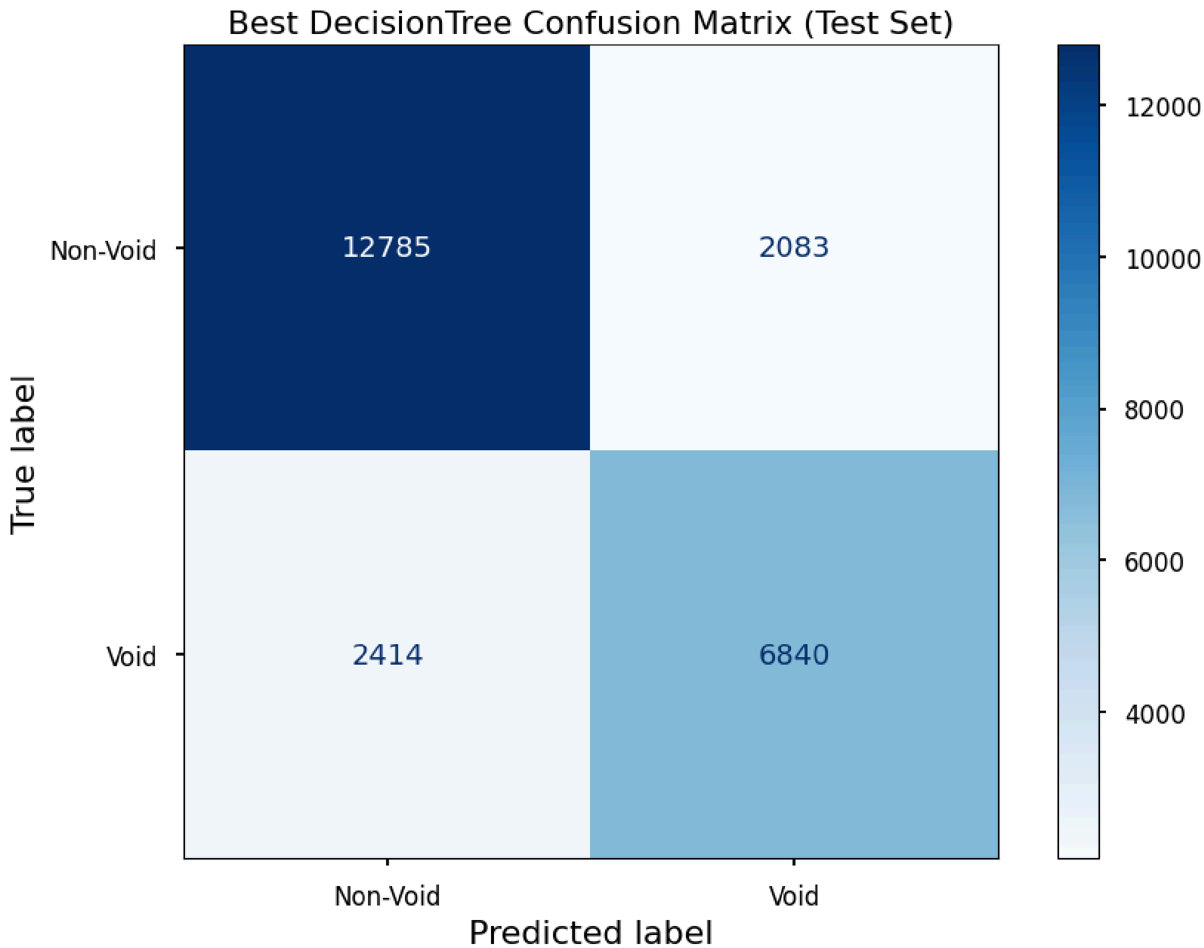
```
Finished fitting LinearSVC in 0.24 seconds.
Best LinearSVC Recall (CV): 0.5000
Best LinearSVC Parameters: {'classifier__C': 10}
--- Refitting best LinearSVC estimator on FULL training data (96484 points)... ---
Finished refitting in 0.05 seconds.

--- Tuning RBFSVC ---
Parameter grid size for RBFSVC: 4
Using HalvingGridSearchCV (grid size 4 <= 20).
Fitting HalvingGridSearchCV for RBFSVC using 66484 samples...
n_iterations: 2
n_required_iterations: 2
n_possible_iterations: 8
min_resources_: 12
max_resources_: 66484
aggressive_elimination: False
factor: 3
----------
iter: 0
n_candidates: 4
n_resources: 12
Fitting 3 folds for each of 4 candidates, totalling 12 fits
----------
iter: 1
n_candidates: 2
n_resources: 36
Fitting 3 folds for each of 2 candidates, totalling 6 fits
Finished fitting RBFSVC in 1087.99 seconds.
Best RBFSVC Recall (CV): 0.5556
Best RBFSVC Parameters: {'classifier__C': 10, 'classifier__gamma': 0.1}
--- Refitting best RBFSVC estimator on FULL training data (96484 points)... ---
Finished refitting in 2472.69 seconds.

--- Tuning NystroemApproxSVC ---
Parameter grid size for NystroemApproxSVC: 27
Using HalvingRandomSearchCV (grid size 27 > 20).
Fitting HalvingRandomSearchCV for NystroemApproxSVC using 66484 samples...
n_iterations: 4
n_required_iterations: 4
n_possible_iterations: 8
min_resources_: 12
max_resources_: 66484
aggressive_elimination: False
factor: 3
----------
iter: 0
n_candidates: 27
n_resources: 12
Fitting 3 folds for each of 27 candidates, totalling 81 fits
----------
iter: 1
n_candidates: 9
n_resources: 36
Fitting 3 folds for each of 9 candidates, totalling 27 fits
----------
iter: 2
```

```
n_candidates: 3
n_resources: 108
Fitting 3 folds for each of 3 candidates, totalling 9 fits
----------
iter: 3
n_candidates: 1
n_resources: 324
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Finished fitting NystroemApproxSVC in 26.87 seconds.
Best NystroemApproxSVC Recall (CV): 0.4380
Best NystroemApproxSVC Parameters: {'classifier__nystroem__n_components': 200, 'clas
sifier__nystroem__gamma': 0.1, 'classifier__linear_svc__C': 10}
--- Refitting best NystroemApproxSVC estimator on FULL training data (96484 point
s)... ---
Finished refitting in 29.69 seconds.

--- Hyperparameter Tuning Summary ---
                      Best Score (Recall)  \
Model
DecisionTree                     0.397436
RandomForest                     0.405681
HistGradientBoosting             0.448718
KNN                              0.277778
LinearSVC                        0.500000
RBFSVC                           0.555556
NystroemApproxSVC                0.438018


Best Params  \
Model
DecisionTree
{'classifier__max_depth': None, 'classifier__min_samples_leaf': 5}
RandomForest          {'classifier__n_estimators': 100, 'classifier__min_samples_lea
f': 5, 'classifier__max_depth': None, 'classifier__class_weight': 'balanced'}
HistGradientBoosting                                              {'classifier__max
_iter': 200, 'classifier__max_depth': None, 'classifier__learning_rate': 0.1}
KNN
{'classifier__n_neighbors': 5, 'classifier__weights': 'distance'}
LinearSVC
{'classifier__C': 10}
RBFSVC
{'classifier__C': 10, 'classifier__gamma': 0.1}
NystroemApproxSVC                                 {'classifier__nystroem__n_component
s': 200, 'classifier__nystroem__gamma': 0.1, 'classifier__linear_svc__C': 10}

                                Search Type  Tuning Time (s)
Model
DecisionTree            HalvingGridSearchCV         4.036848
RandomForest          HalvingRandomSearchCV        15.387128
HistGradientBoosting  HalvingRandomSearchCV         1.463760
KNN                     HalvingGridSearchCV         0.518918
LinearSVC               HalvingGridSearchCV         0.236827
RBFSVC                  HalvingGridSearchCV      1087.986660
NystroemApproxSVC     HalvingRandomSearchCV        26.874744
```

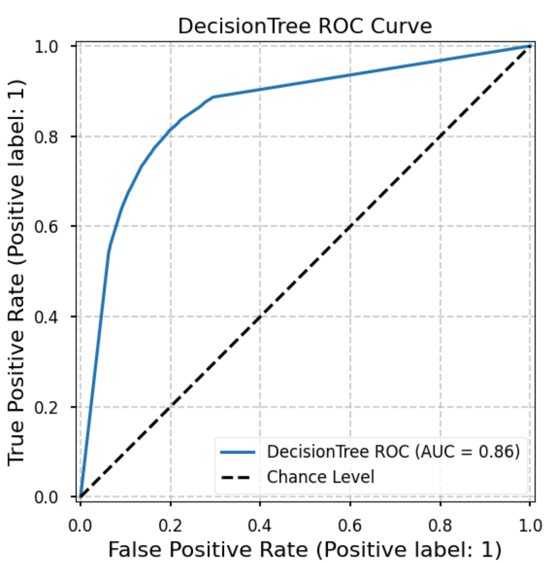
In [11]:
```
# --- 11. Final Model Evaluation ---
print("\n--- Evaluating Tuned Models on Test Set ---")

if 'best_estimators' in locals() and 'X_test' in locals():
    evaluation_results = []

    for name, model in best_estimators.items():
        print(f"\n--- Evaluating Best {name} ---")
        try:
            # Make predictions on the test set
            y_pred = model.predict(X_test)
            # Get probabilities for ROC/PR curves if possible
            try:
                y_pred_proba = model.predict_proba(X_test)[:, 1] # Probability of c
                has_proba = True
            except AttributeError:
                print(f"Note: {name} does not support predict_proba, skipping ROC/P
                y_pred_proba = None
                has_proba = False


            # Calculate metrics
            accuracy = accuracy_score(y_test, y_pred)
            precision = precision_score(y_test, y_pred, pos_label=1)
            recall = recall_score(y_test, y_pred, pos_label=1)
            f1 = f1_score(y_test, y_pred, pos_label=1)

            evaluation_results.append({
                'Model': name,
                'Accuracy': accuracy,
                'Precision (Void)': precision,
                'Recall (Void)': recall,
                'F1 (Void)': f1
            })

            # Display Confusion Matrix
            cm = confusion_matrix(y_test, y_pred)
            disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=['Non
            disp.plot(cmap=plt.cm.Blues)
            plt.title(f'Best {name} Confusion Matrix (Test Set)')
            plt.show()

            # Display Classification Report
            print("\nClassification Report:")
            print(classification_report(y_test, y_pred, target_names=['Non-Void (0)

            # Plot ROC Curve (if probabilities are available)
            if has_proba:
                fig, ax = plt.subplots(1, 2, figsize=(14, 6))
                RocCurveDisplay.from_predictions(y_test, y_pred_proba, name=f'{name
                ax[0].plot([0, 1], [0, 1], 'k--', label='Chance Level')
                ax[0].set_title(f'{name} ROC Curve')
                ax[0].legend()
                ax[0].grid(True, linestyle='--', alpha=0.6)
```

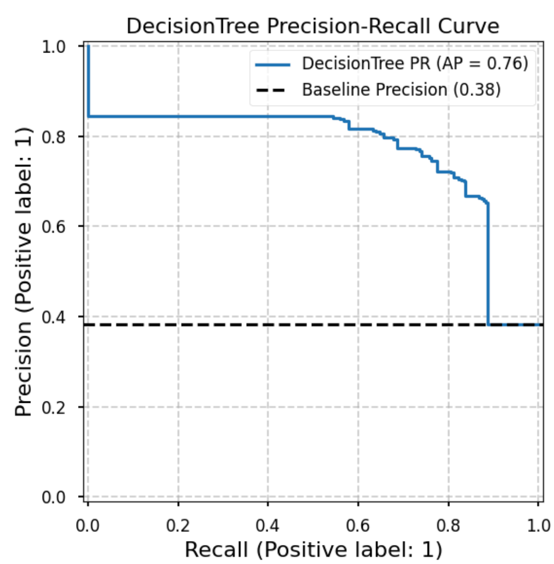
```python
                # Plot Precision-Recall Curve
                PrecisionRecallDisplay.from_predictions(y_test, y_pred_proba, name=
                # Calculate baseline precision (fraction of positive class)
                baseline_precision = np.sum(y_test == 1) / len(y_test)
                ax[1].axhline(baseline_precision, color='k', linestyle='--', label=
                ax[1].set_title(f'{name} Precision-Recall Curve')
                ax[1].legend()
                ax[1].grid(True, linestyle='--', alpha=0.6)

                plt.tight_layout()
                plt.show()


        except Exception as e:
            print(f"An error occurred during evaluation of {name}: {e}")


    # Summarize results in a DataFrame
    print("\n--- Final Model Performance Summary (Test Set) ---")
    evaluation_df = pd.DataFrame(evaluation_results)
    # Sort by desired metric, e.g., Recall or F1
    evaluation_df = evaluation_df.sort_values(by='Recall (Void)', ascending=False)
    print(evaluation_df.round(4)) # Round for readability

    # --- Select the final best model ---
    final_model_name = evaluation_df.iloc[0]['Model']
    final_model = best_estimators[final_model_name]
    print(f"\nSelected final model: {final_model_name}")


else:
    print("Skipping final evaluation because best estimators or test data are missi
    final_model = None # Ensure final_model is defined
```

```
--- Evaluating Tuned Models on Test Set ---

--- Evaluating Best DecisionTree ---
```

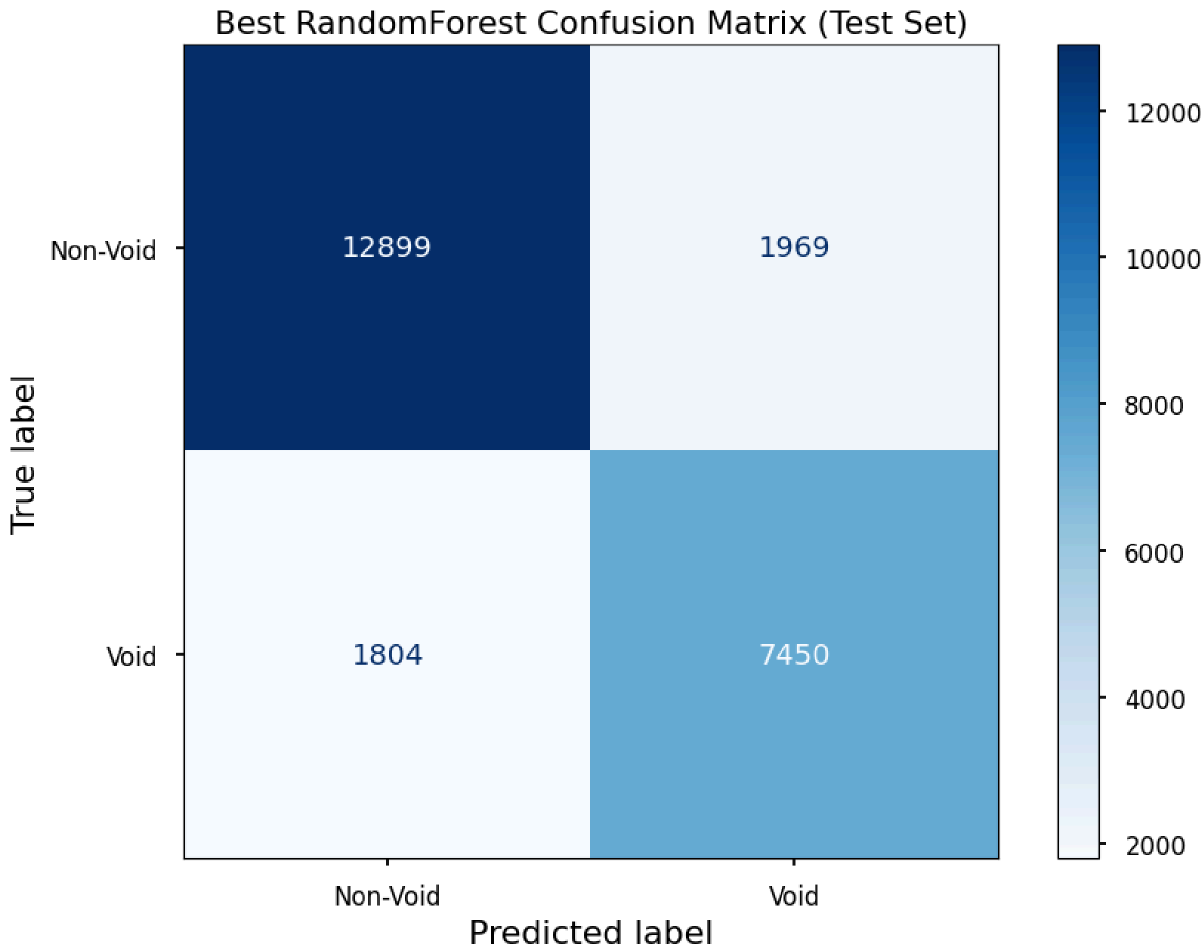



```
Classification Report:
              precision    recall  f1-score   support

Non-Void (0)       0.84      0.86      0.85     14868
    Void (1)       0.77      0.74      0.75      9254

    accuracy                           0.81     24122
   macro avg       0.80      0.80      0.80     24122
weighted avg       0.81      0.81      0.81     24122
```

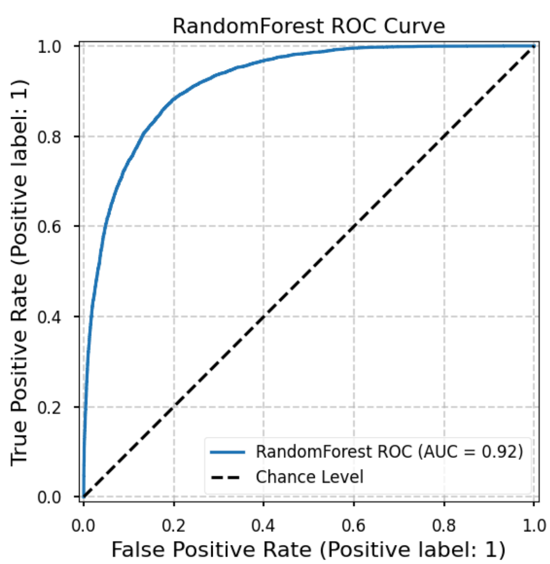


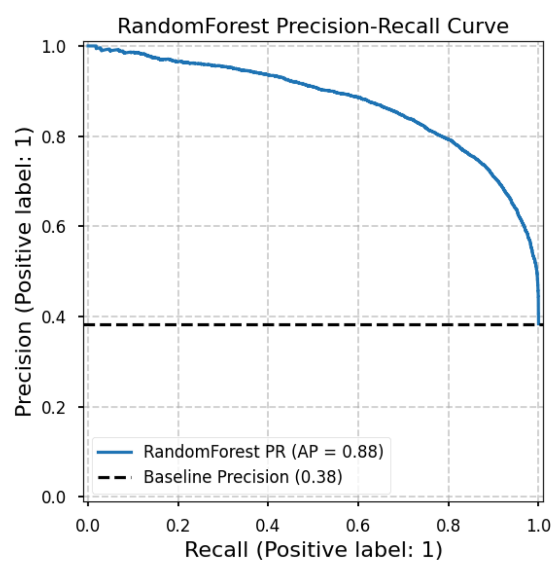



```
--- Evaluating Best RandomForest ---
```

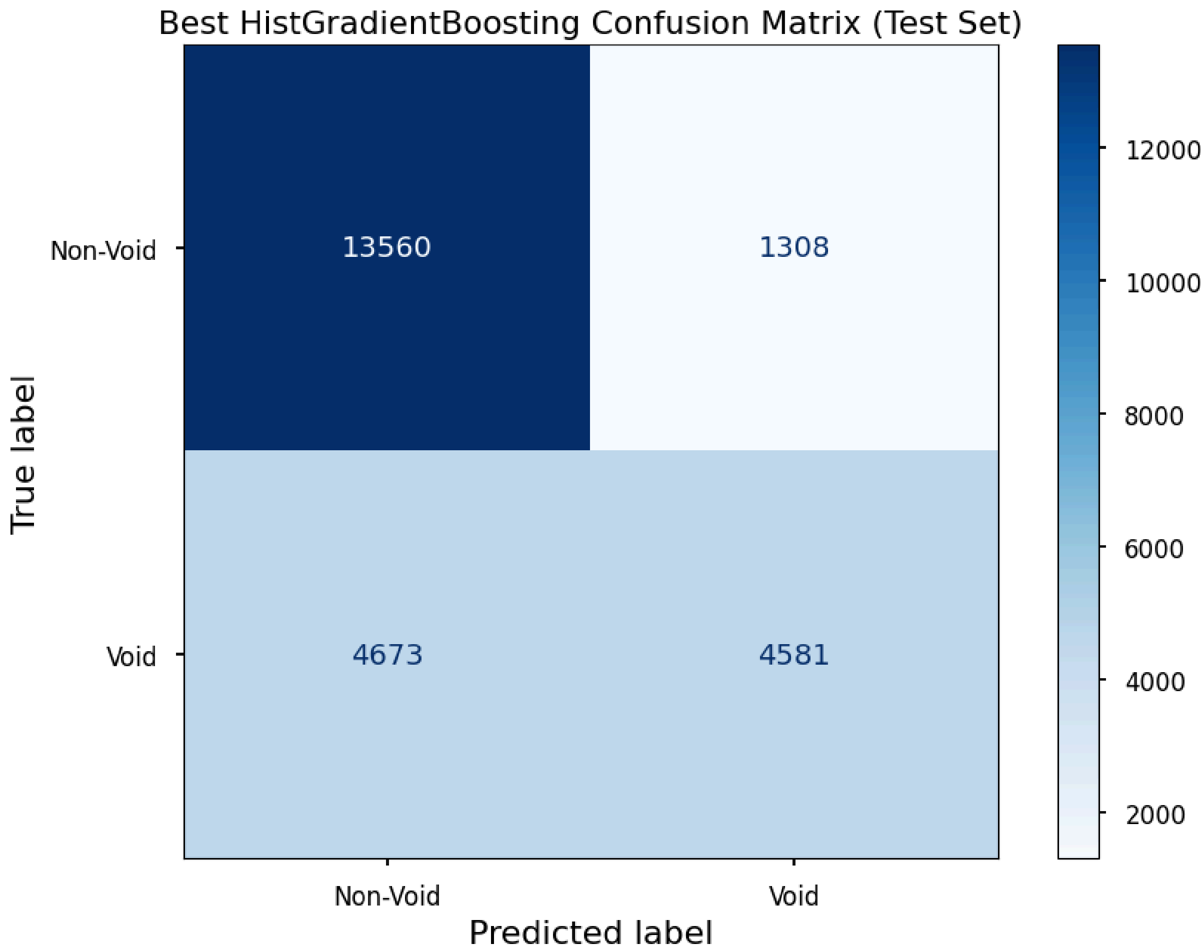



```
Classification Report:
              precision    recall  f1-score   support

Non-Void (0)       0.88      0.87      0.87     14868
    Void (1)       0.79      0.81      0.80      9254

    accuracy                           0.84     24122
   macro avg       0.83      0.84      0.84     24122
weighted avg       0.84      0.84      0.84     24122
```

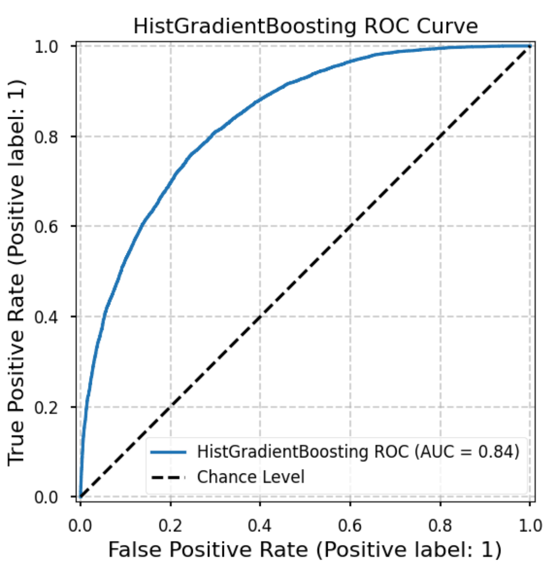


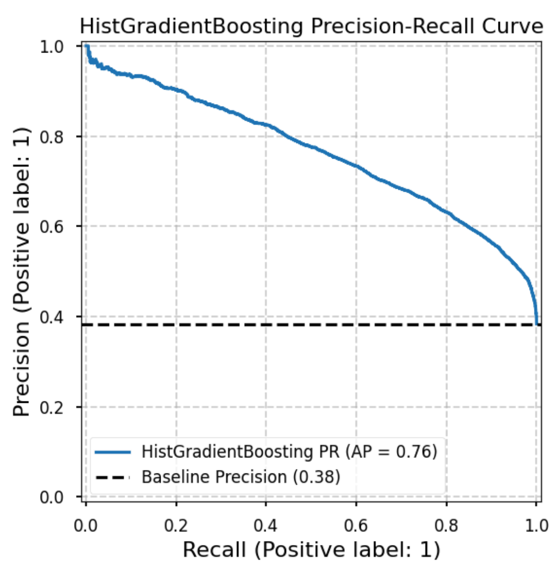



```
--- Evaluating Best HistGradientBoosting ---
```

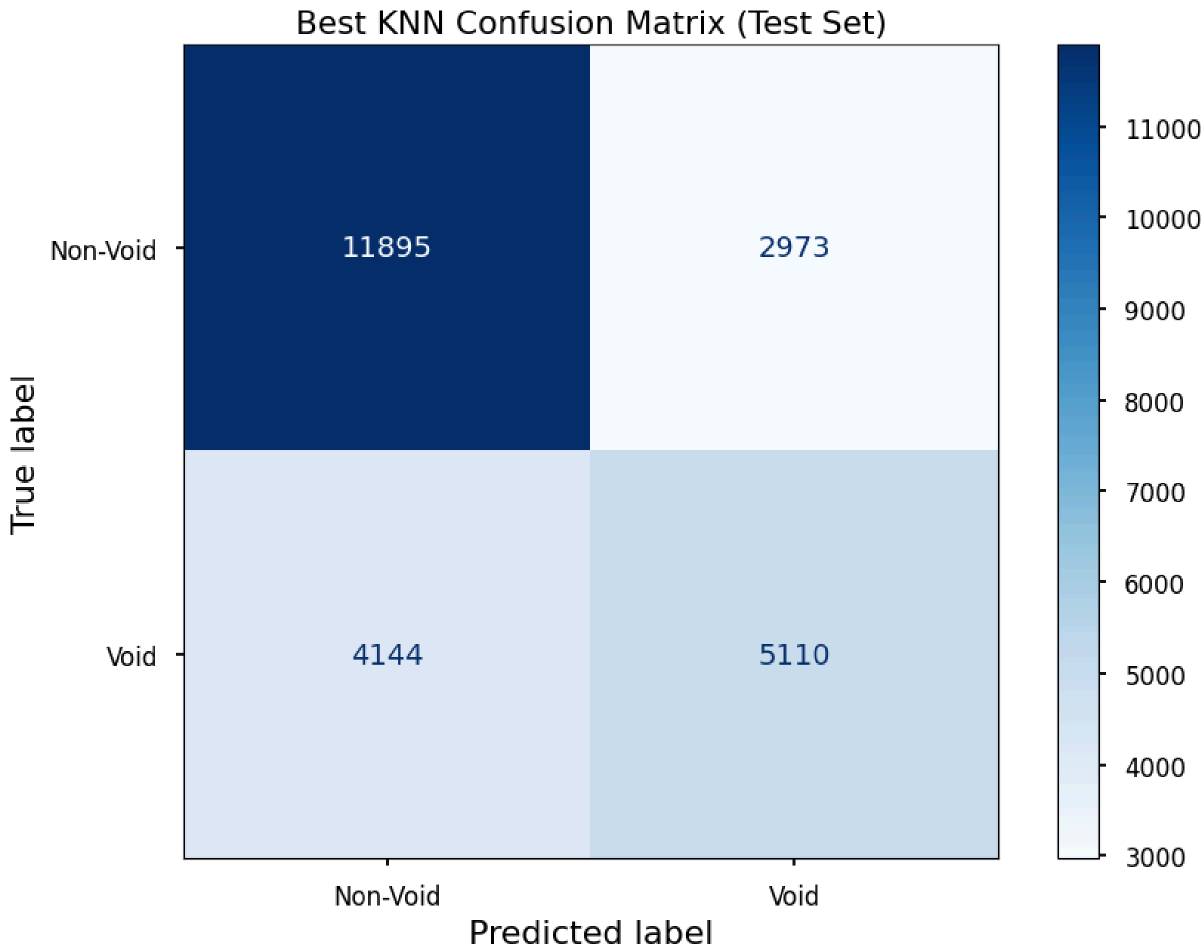



```
Classification Report:
              precision    recall  f1-score   support

Non-Void (0)       0.74      0.91      0.82     14868
    Void (1)       0.78      0.50      0.61      9254

    accuracy                           0.75     24122
   macro avg       0.76      0.70      0.71     24122
weighted avg       0.76      0.75      0.74     24122
```

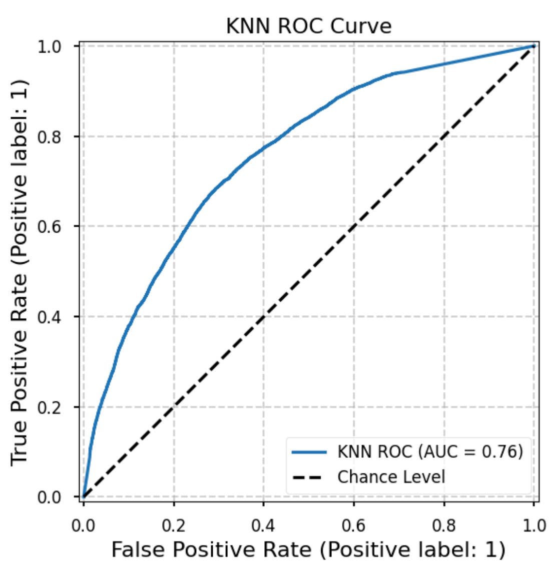


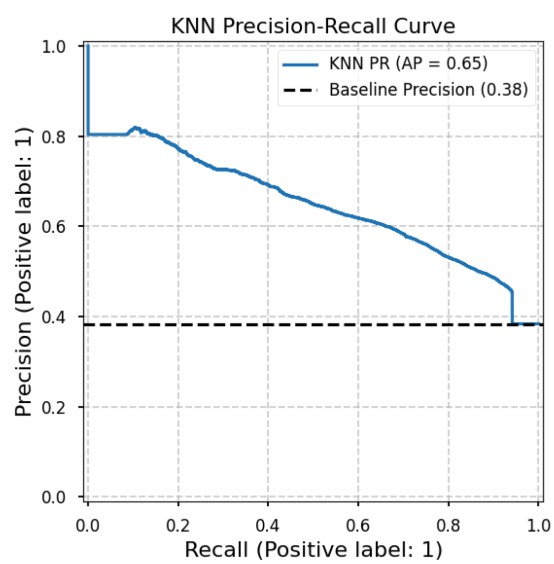



```
--- Evaluating Best KNN ---
```

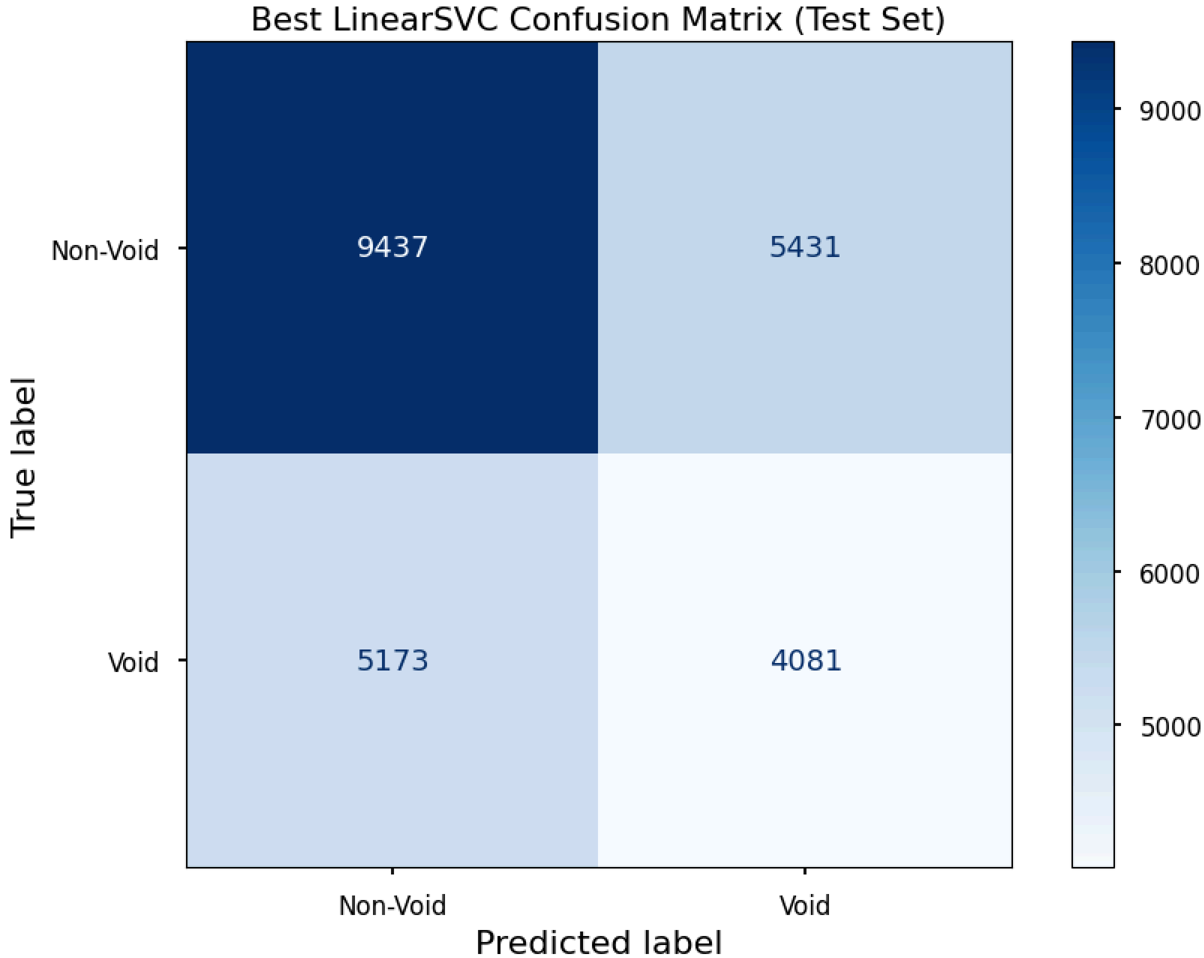



```
Classification Report:
              precision    recall  f1-score   support

Non-Void (0)       0.74      0.80      0.77     14868
    Void (1)       0.63      0.55      0.59      9254

    accuracy                           0.70     24122
   macro avg       0.69      0.68      0.68     24122
weighted avg       0.70      0.70      0.70     24122
```

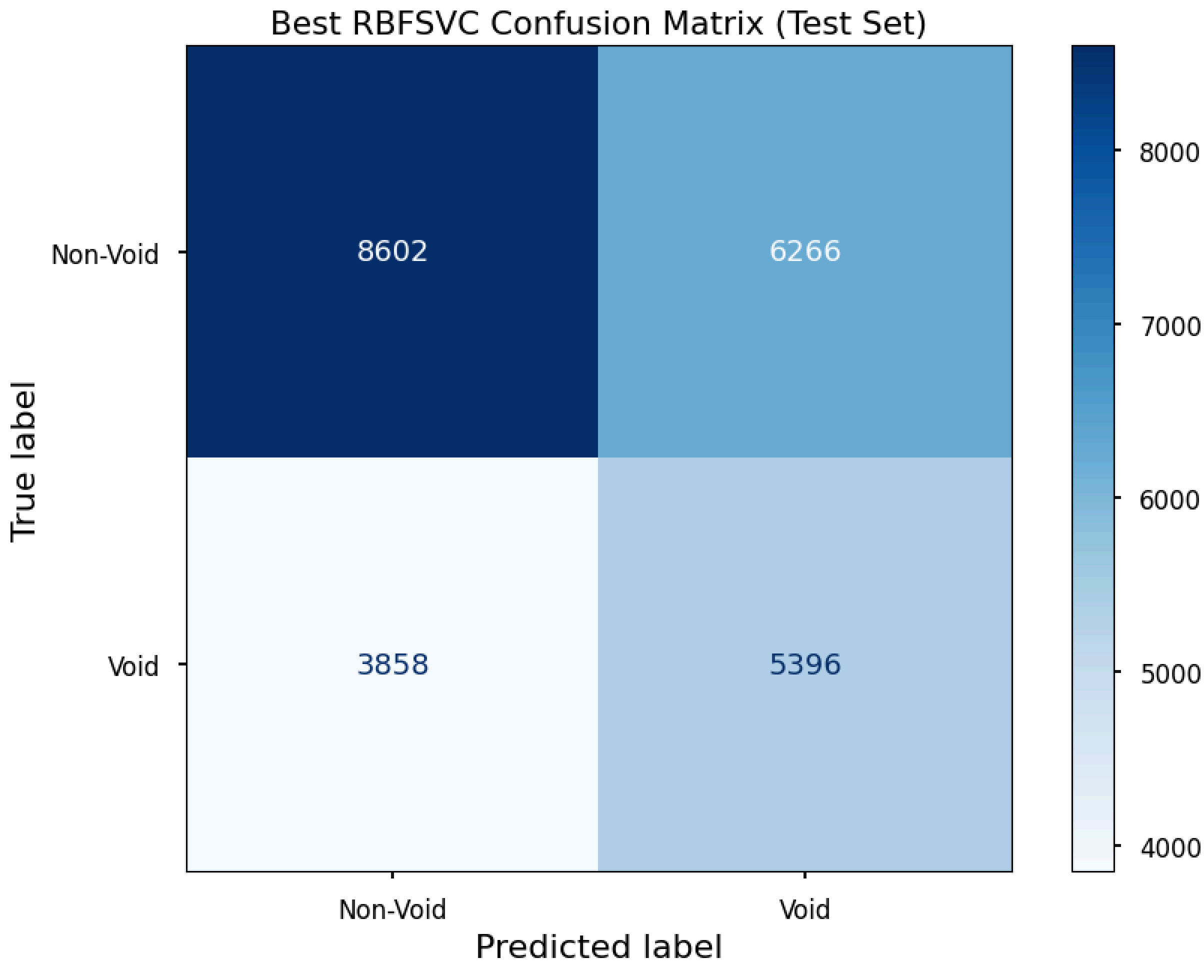




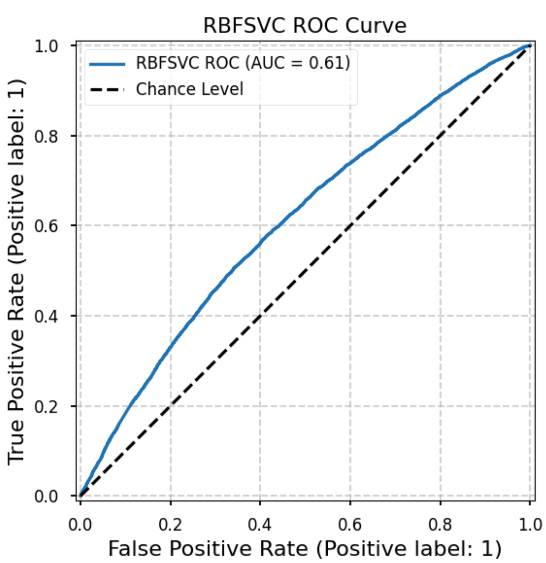
```
--- Evaluating Best LinearSVC ---
Note: LinearSVC does not support predict_proba, skipping ROC/PR curves.
```

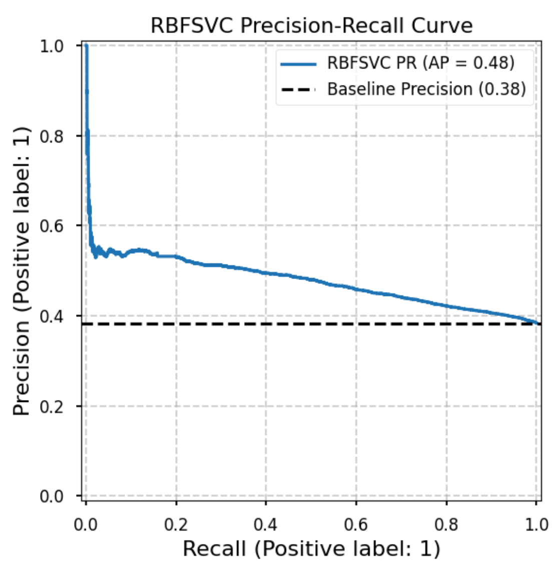



```
Classification Report:
              precision    recall  f1-score   support

Non-Void (0)       0.65      0.63      0.64     14868
    Void (1)       0.43      0.44      0.43      9254

    accuracy                           0.56     24122
   macro avg       0.54      0.54      0.54     24122
weighted avg       0.56      0.56      0.56     24122


--- Evaluating Best RBFSVC ---
```

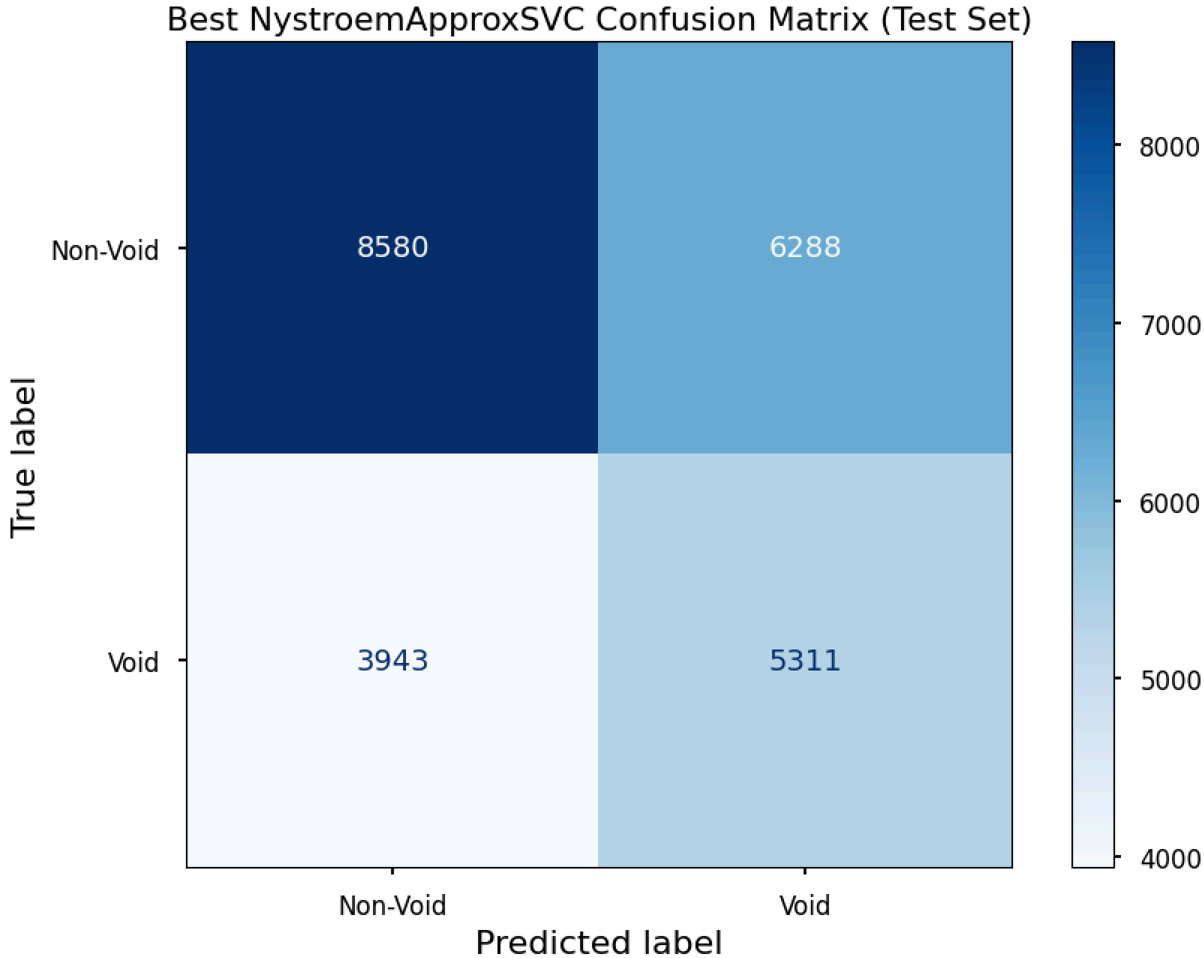



```
Classification Report:
              precision    recall  f1-score   support

Non-Void (0)       0.69      0.58      0.63     14868
    Void (1)       0.46      0.58      0.52      9254

    accuracy                           0.58     24122
   macro avg       0.58      0.58      0.57     24122
weighted avg       0.60      0.58      0.59     24122
```





```
--- Evaluating Best NystroemApproxSVC ---
Note: NystroemApproxSVC does not support predict_proba, skipping ROC/PR curves.
```



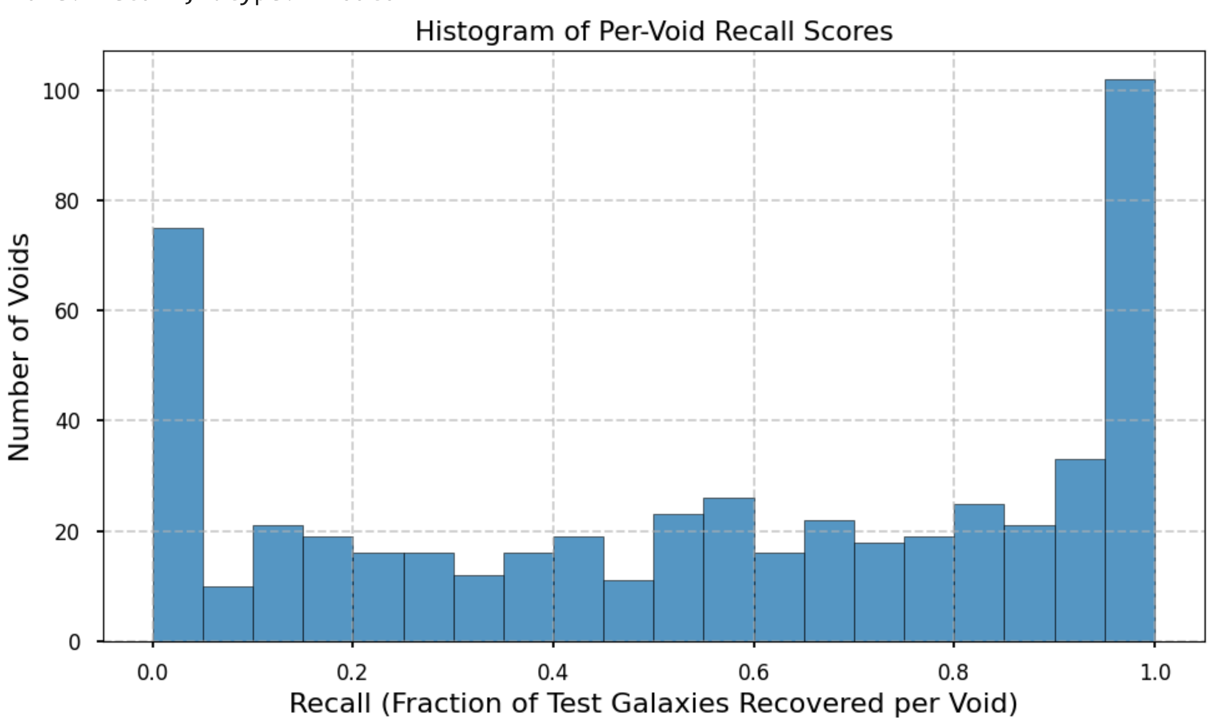

```
Classification Report:
              precision    recall  f1-score   support

Non-Void (0)       0.69      0.58      0.63     14868
    Void (1)       0.46      0.57      0.51      9254

    accuracy                           0.58     24122
   macro avg       0.57      0.58      0.57     24122
weighted avg       0.60      0.58      0.58     24122


--- Final Model Performance Summary (Test Set) ---
                  Model  Accuracy  Precision (Void)  Recall (Void)  F1 (Void)
1          RandomForest    0.8436            0.7910         0.8051     0.7979
0          DecisionTree    0.8136            0.7666         0.7391     0.7526
5                RBFSVC    0.5803            0.4627         0.5831     0.5160
6     NystroemApproxSVC    0.5759            0.4579         0.5739     0.5094
3                   KNN    0.7050            0.6322         0.5522     0.5895
2  HistGradientBoosting    0.7521            0.7779         0.4950     0.6050
4             LinearSVC    0.5604            0.4290         0.4410     0.4349

Selected final model: RandomForest
```

In [12]:
```
# --- 11b. Ground-Truth Comparison & Per-Void Recovery ---
print("\n--- Ground-Truth Comparison & Per-Void Recovery Analysis ---")
```

```
# Ensure we have the necessary data:
# - y_test: True labels for the test set galaxies
# - y_pred: Predictions from the chosen final model for the test set galaxies
# - galaxies_df: Original DataFrame with galaxy info, including an index that align
# - voids_df: DataFrame with void info (center coords, radius)
# - X_test_indices: The original indices from galaxies_df corresponding to X_test/y

if 'final_model' in locals() and final_model is not None and 'X_test' in locals() a
    try:
        # Get predictions from the final chosen model
        if 'y_pred' not in locals() or len(y_pred) != len(y_test):
             print(f"Generating predictions using the final model: {final_model_nam
             y_pred = final_model.predict(X_test)

        # 1. Standard Classification Metrics (already printed in Step 11, but repea
        print("\nOverall Test Set Performance (Final Model):")
        cm_final = confusion_matrix(y_test, y_pred)
        print("Confusion Matrix:")
        print(cm_final)
        print("\nClassification Report:")
        print(classification_report(y_test, y_pred, target_names=['Non-Void (0)', '
        final_recall = recall_score(y_test, y_pred, pos_label=1) # Overall recall

        # --- Per-Void Recovery Calculation ---
        print("\nCalculating Per-Void Recovery...")

        # We need to map the test set predictions back to the original galaxy indic
        # This assumes train_test_split kept the original index if X was a DataFram
        # or we retrieve the indices from the split. Let's assume we have the indic
        # If X was created directly from galaxies_df[feature_cols].values, the indi
        # Re-run train_test_split with indices if necessary:
        _, _, _, _, idx_train, idx_test = train_test_split(
            X, y, galaxies_df.index, # Include index here
            test_size=test_size,
            random_state=random_state,
            stratify=y
        )

        # Create a temporary DataFrame for test set galaxies with their true/predic
        test_galaxies_df = galaxies_df.loc[idx_test].copy()
        test_galaxies_df['true_is_void'] = y_test # Corresponds to idx_test order
        test_galaxies_df['pred_is_void'] = y_pred # Corresponds to idx_test order

        per_void_recall = []
        processed_void_indices = []
        num_test_galaxies_in_any_void = 0

        # Iterate through each *ground-truth* void
        for void_idx, void_row in voids_df.iterrows():
            # Find TRUE void galaxies (from the original labeling) that belong to *
            # Note: A galaxy is labeled 'is_void' if it's inside its *nearest* void
            # We need galaxies in the test set whose *true* nearest void was this o
            true_members_in_test = test_galaxies_df[
                (test_galaxies_df['nearest_void_idx'] == void_idx) &
                (test_galaxies_df['true_is_void'] == 1) # Ensure they are truly voi
```

```
            ]

            if not true_members_in_test.empty:
                num_test_galaxies_in_this_void = len(true_members_in_test)
                num_test_galaxies_in_any_void += num_test_galaxies_in_this_void

                # Count how many of these were correctly predicted as void galaxies
                correctly_predicted = true_members_in_test['pred_is_void'].sum() #

                # Calculate recall for this specific void
                recall_this_void = correctly_predicted / num_test_galaxies_in_this_
                per_void_recall.append(recall_this_void)
                processed_void_indices.append(void_idx)

        # Create a DataFrame for per-void results
        per_void_df = pd.DataFrame({
            'void_index': processed_void_indices,
            'recall': per_void_recall
        })

        # 3. Summarize per-void recall statistics
        print("\nPer-Void Recall Statistics (for voids with galaxies in the test se
        print(per_void_df['recall'].describe())

        # Plot histogram of per-void recall scores
        plt.figure(figsize=(10, 6))
        sns.histplot(per_void_df['recall'], bins=20, kde=False)
        plt.title('Histogram of Per-Void Recall Scores')
        plt.xlabel('Recall (Fraction of Test Galaxies Recovered per Void)')
        plt.ylabel('Number of Voids')
        plt.grid(True, linestyle='--', alpha=0.6)
        plt.tight_layout()
        plt.show()

        # 4. Print overall summary
        mean_recall = per_void_df['recall'].mean()
        std_recall = per_void_df['recall'].std()
        num_voids_in_test = len(per_void_df)
        total_voids = len(voids_df)

        print("\n--- Per-Void Recovery Summary ---")
        print(f"Analyzed {num_voids_in_test} out of {total_voids} total voids that
        print(f"Number of test-set galaxies truly belonging to these voids: {num_te
        print(f"Overall test set recall (Void class): {final_recall:.3f}")
        print(f"Mean per-void recall: {mean_recall:.3f} (i.e., on average, recovere
        print(f"Standard deviation of per-void recall: {std_recall:.3f}")
        print(f"Median per-void recall: {per_void_df['recall'].median():.3f}")


    except Exception as e:
        print(f"An error occurred during per-void recovery analysis: {e}")
else:
    print("Skipping per-void recovery analysis because final model, test data, or o
```

```
--- Ground-Truth Comparison & Per-Void Recovery Analysis ---

Overall Test Set Performance (Final Model):
Confusion Matrix:
[[8580 6288]
 [3943 5311]]

Classification Report:
              precision    recall  f1-score   support

Non-Void (0)       0.69      0.58      0.63     14868
    Void (1)       0.46      0.57      0.51      9254

    accuracy                           0.58     24122
   macro avg       0.57      0.58      0.57     24122
weighted avg       0.60      0.58      0.58     24122


Calculating Per-Void Recovery...

Per-Void Recall Statistics (for voids with galaxies in the test set):
count    520.000000
mean       0.553776
std        0.358196
min        0.000000
25%        0.211722
50%        0.600000
75%        0.900000
max        1.000000
Name: recall, dtype: float64
```

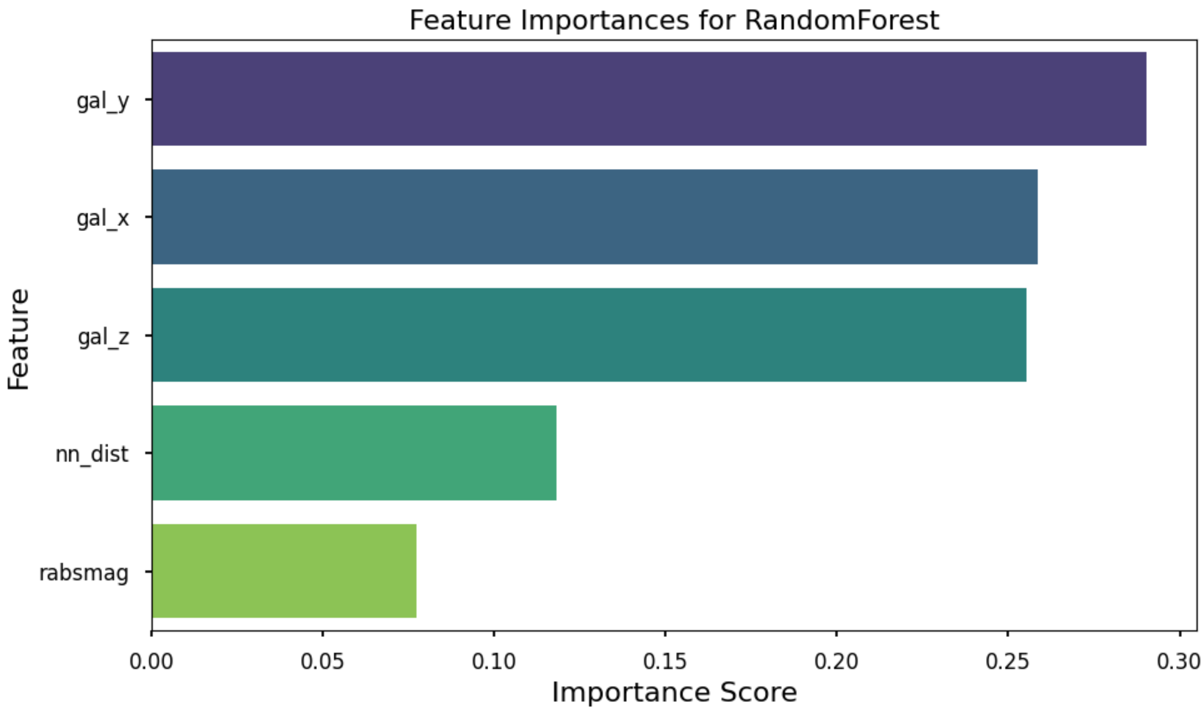


```
--- Per-Void Recovery Summary ---
Analyzed 520 out of 531 total voids that had associated galaxies in the test set.
Number of test-set galaxies truly belonging to these voids: 9254
Overall test set recall (Void class): 0.574
Mean per-void recall: 0.554 (i.e., on average, recovered 55.4% of test galaxies per
void)
Standard deviation of per-void recall: 0.358
Median per-void recall: 0.600
```

In [13]:
```python
# --- 12. Feature Importance & Interpretation ---
print("\n--- Analyzing Feature Importance ---")

# Use the selected final model
if 'final_model' in locals() and final_model is not None:
    # Check if the final model's classifier step has feature_importances_
    try:
        # Access the classifier step within the pipeline
        classifier_step = final_model.named_steps['classifier']
        feature_names = feature_cols # From Step 8

        if hasattr(classifier_step, 'feature_importances_'):
            print(f"Extracting feature importances from final model ({final_model_n
            importances = classifier_step.feature_importances_

            # Create DataFrame for plotting
            importance_df = pd.DataFrame({'Feature': feature_names, 'Importance': i
            importance_df = importance_df.sort_values(by='Importance', ascending=Fa

            # Plot feature importances
            plt.figure(figsize=(10, 6))
            sns.barplot(x='Importance', y='Feature', data=importance_df, palette='v
            plt.title(f'Feature Importances for {final_model_name}')
            plt.xlabel('Importance Score')
            plt.ylabel('Feature')
            plt.tight_layout()
            plt.show()

            print("\nFeature Importances:")
            print(importance_df)

        else:
            print(f"The selected model ({final_model_name}) does not have a 'featur
            print("Consider using permutation importance for model-agnostic insight

            # --- Permutation Importance  ---
            print("\nCalculating Permutation Importance (can take time)...")
            perm_importance = permutation_importance(
                final_model, X_test, y_test, n_repeats=10, random_state=random_stat
                scoring=make_scorer(recall_score, pos_label=1) # Score based on rec
            )
            sorted_idx = perm_importance.importances_mean.argsort()[::-1] # Sort de
            perm_importance_df = pd.DataFrame({
                'Feature': np.array(feature_names)[sorted_idx],
                'Importance Mean': perm_importance.importances_mean[sorted_idx],
                'Importance Std': perm_importance.importances_std[sorted_idx]
            })
```

```
            plt.figure(figsize=(10, 6))
            sns.barplot(x='Importance Mean', y='Feature', data=perm_importance_df,
            plt.title(f'Permutation Importances for {final_model_name} (Scored by R
            plt.xlabel('Mean Importance (Drop in Recall)')
            plt.ylabel('Feature')
            plt.tight_layout()
            plt.show()
            print("\nPermutation Importances:")
            print(perm_importance_df)


    except Exception as e:
        print(f"An error occurred during feature importance analysis: {e}")
else:
    print("Skipping feature importance analysis because the final model is not avai
```

```
--- Analyzing Feature Importance ---
Extracting feature importances from final model (RandomForest)...
```



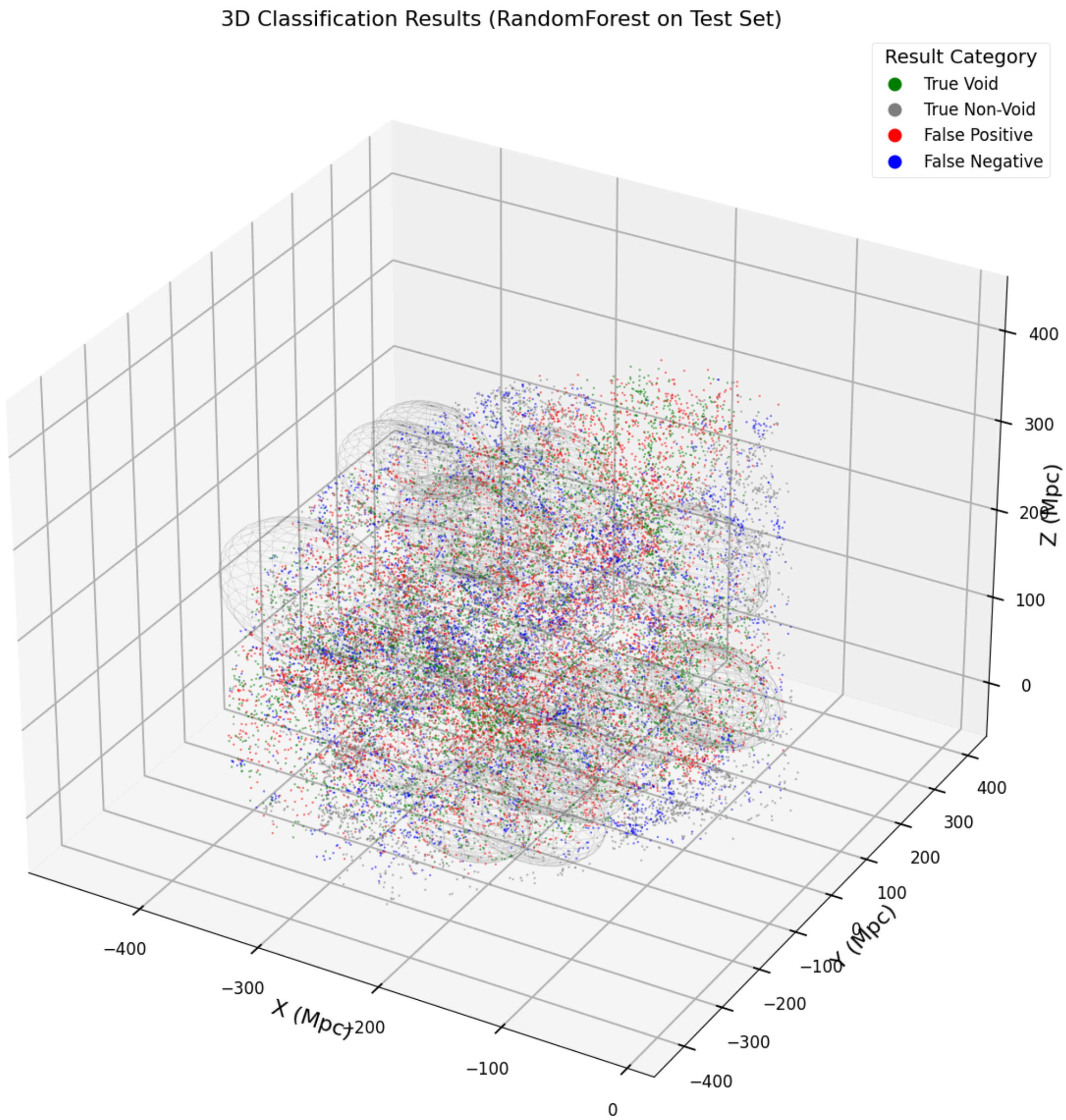

```
Feature Importances:
   Feature  Importance
1    gal_y    0.290479
0    gal_x    0.258709
2    gal_z    0.255391
4  nn_dist    0.118181
3  rabsmag    0.077240
```

In [14]:
```
# --- 13. 3D Visualization of Results ---
print("\n--- Creating 3D Visualization of Classification Results ---")

# Requires test set coordinates, true labels (y_test), predicted labels (y_pred)
# And potentially void data for overlays

if 'X_test' in locals() and 'y_test' in locals() and 'y_pred' in locals() and 'fina
    try:
```

```
# Get the test set coordinates (unscaled)
# Assuming idx_test is available from Step 11b or re-run split
if 'idx_test' not in locals():
     _, _, _, _, _, idx_test = train_test_split(
        X, y, galaxies_df.index, test_size=test_size, random_state=random_s

test_coords_df = galaxies_df.loc[idx_test, ['gal_x', 'gal_y', 'gal_z']].cop
test_coords_df['true_label'] = y_test
test_coords_df['pred_label'] = y_pred

# Define categories: TP, TN, FP, FN
# Void is positive (1), Non-Void is negative (0)
conditions = [
    (test_coords_df['true_label'] == 1) & (test_coords_df['pred_label'] ==
    (test_coords_df['true_label'] == 0) & (test_coords_df['pred_label'] ==
    (test_coords_df['true_label'] == 0) & (test_coords_df['pred_label'] ==
    (test_coords_df['true_label'] == 1) & (test_coords_df['pred_label'] ==
]
categories = ['True Void', 'True Non-Void', 'False Positive', 'False Negati
colors = ['green', 'gray', 'red', 'blue'] # TP, TN, FP, FN
test_coords_df['result_category'] = np.select(conditions, categories, defau
category_colors = dict(zip(categories, colors))

# --- Create 3D Plot ---
print("Generating 3D scatter plot of test set results...")
# Subsample if the test set is very large
plot_fraction = 0.2
if len(test_coords_df) * plot_fraction > 10000:
     plot_df = test_coords_df.sample(frac=plot_fraction, random_state=rando
     print(f"Plotting a {plot_fraction*100:.0f}% subsample ({len(plot_df)}
else:
     plot_df = test_coords_df
     print(f"Plotting all {len(plot_df)} test points.")


fig = plt.figure(figsize=(14, 12))
ax = fig.add_subplot(111, projection='3d')

# Scatter plot colored by result category
scatter = ax.scatter(plot_df['gal_x'], plot_df['gal_y'], plot_df['gal_z'],
                     c=plot_df['result_category'].map(category_colors),
                     s=5, alpha=0.5, marker='.')

ax.set_title(f'3D Classification Results ({final_model_name} on Test Set)')
ax.set_xlabel('X (Mpc)')
ax.set_ylabel('Y (Mpc)')
ax.set_zlabel('Z (Mpc)')

# Create custom legend
legend_elements = [Line2D([0], [0], marker='o', color='w', label=cat, marke
                   for cat, col in category_colors.items()]
ax.legend(handles=legend_elements, title="Result Category")
ax.grid(True)


if 'voids_df' in locals():
```

```python
            print("Overlaying void spheres (subset for clarity)...")
            n_voids_to_plot = min(20, len(voids_df)) # Plot up to 20 voids
            voids_to_plot = voids_df.nlargest(n_voids_to_plot, 'void_radius_mpc')

            for _, void_row in voids_to_plot.iterrows():
                # Draw sphere wireframe
                u_sphere = np.linspace(0, 2 * np.pi, 20)
                v_sphere = np.linspace(0, np.pi, 20)
                x_sphere = void_row['void_x'] + void_row['void_radius_mpc'] * np.c
                y_sphere = void_row['void_y'] + void_row['void_radius_mpc'] * np.c
                z_sphere = void_row['void_z'] + void_row['void_radius_mpc'] * np.c
                ax.plot_wireframe(x_sphere, y_sphere, z_sphere, color='black', alp

            print(f"Overlayed wireframes for {len(voids_to_plot)} largest voids.")

        plt.tight_layout()
        plt.show()

    except Exception as e:
        print(f"An error occurred during 3D visualization: {e}")
else:
    print("Skipping 3D visualization due to missing data (test coordinates, labels,
```

```
--- Creating 3D Visualization of Classification Results ---
Generating 3D scatter plot of test set results...
Plotting all 24122 test points.
Overlaying void spheres (subset for clarity)...
Overlayed wireframes for 20 largest voids.
```



In [15]:
```
# --- 14. Save Model & Predictions ---
print("\n--- Saving Final Model and Exporting Predictions ---")

if 'final_model' in locals() and final_model is not None and 'galaxies_df' in local
    try:
        # 1. Save the final chosen model pipeline
        model_filename = f'final_void_classifier_{final_model_name}.joblib'
        joblib.dump(final_model, model_filename)
        print(f"Final model saved successfully to: {model_filename}")

        # 2. Add predictions for *all* galaxies to the original DataFrame
        print("Generating predictions for the *entire* galaxy dataset...")
        all_predictions = final_model.predict(galaxies_df[feature_cols].values)
        galaxies_df['pred_is_void'] = all_predictions.astype(int) # Add as integer

        print("Added 'pred_is_void' column to the main galaxy DataFrame.")
        print(galaxies_df[['is_void', 'pred_is_void']].head()) # Show true vs predi
        print("\nValue counts for predictions on full dataset:")
```

```
        print(galaxies_df['pred_is_void'].value_counts(dropna=False))

        # 3. Export the DataFrame to CSV
        output_csv_filename = 'galaxies_with_predictions.csv'
        columns_to_export = [
            'ra', 'dec', 'redshift', 'Rgal', 'Rgal_Mpc', 'rabsmag', # Original + de
            'gal_x', 'gal_y', 'gal_z', # Cartesian coords
            'nn_dist', # Engineered feature
            'dist_to_nearest_void', 'nearest_void_idx', 'radius_of_nearest_void', #
            'is_void', # True label (ground truth)
            'pred_is_void' # Model prediction
        ]
        # Filter out columns that might not exist if steps failed
        columns_to_export = [col for col in columns_to_export if col in galaxies_df

        galaxies_df[columns_to_export].to_csv(output_csv_filename, index=False)
        print(f"Galaxy DataFrame with predictions exported successfully to: {output
        print(f"Exported columns: {columns_to_export}")

    except Exception as e:
        print(f"An error occurred during saving/exporting: {e}")
else:
    print("Skipping model saving and prediction export because the final model or g
```

```
--- Saving Final Model and Exporting Predictions ---
Final model saved successfully to: final_void_classifier_RandomForest.joblib
Generating predictions for the *entire* galaxy dataset...
Added 'pred_is_void' column to the main galaxy DataFrame.
   is_void  pred_is_void
0    False             0
1    False             0
2    False             0
3    False             0
4    False             0

Value counts for predictions on full dataset:
pred_is_void
0    73069
1    47537
Name: count, dtype: int64
Galaxy DataFrame with predictions exported successfully to: galaxies_with_prediction
s.csv
Exported columns: ['ra', 'dec', 'redshift', 'Rgal', 'Rgal_Mpc', 'rabsmag', 'gal_x',
'gal_y', 'gal_z', 'nn_dist', 'dist_to_nearest_void', 'nearest_void_idx', 'radius_of_
nearest_void', 'is_void', 'pred_is_void']
```