skubisz_sebastian_finalproject

November 16, 2025

CS634 Final Project *Performance Evaluation of Classification Models for Water Quality Prediction Using SMOTE and Cross-Validation*

Student: Sebastian Skubisz UCID: ss365 Instructor: Dr. yasser

This project aims to preprocess and analyze a water quality dataset to predict whether water is safe to drink using machine learning models such as K-Nearest Neighbors (KNN), Random Forest (RF), and LSTM. The workflow includes data cleaning, visualizing feature distributions, and applying SMOTE to balance the classes.

Hyperparameters for KNN and RF are tuned via grid search, and the models are evaluated using 10-fold cross-validation. The best-performing models are then trained on the entire dataset and tested on a separate test set. Their performance is compared using ROC curves and AUC scores, providing a comprehensive assessment of their prediction capabilities.

[1]:
```
#!/usr/bin/env python3
import pandas as pd
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
warnings.filterwarnings("ignore")
```

This code imports essential libraries for data manipulation and visualization. pandas is used for handling structured data, while numpy supports numerical operations. The matplotlib library is configured with a non-GUI backend to prevent errors in non-interactive environments, and seaborn enhances visualizations. Additionally, warnings are suppressed to maintain a clean output during execution.

[2]:
```
from collections import Counter

from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
from sklearn.metrics import roc_auc_score, roc_curve
from sklearn.metrics import brier_score_loss
```

```
from sklearn.metrics import auc

from keras.models import Sequential
from keras.layers import Dense, LSTM

from imblearn.over_sampling import SMOTE  # SMOTE integration
```

```
2025-11-16 22:01:51.780908: I external/local_xla/xla/tsl/cuda/cudart_stub.cc:31]
Could not find cuda drivers on your machine, GPU will not be used.
2025-11-16 22:01:51.781080: I tensorflow/core/util/port.cc:153] oneDNN custom
operations are on. You may see slightly different numerical results due to
floating-point round-off errors from different computation orders. To turn them
off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.
2025-11-16 22:01:51.809010: I tensorflow/core/platform/cpu_feature_guard.cc:210]
This TensorFlow binary is optimized to use available CPU instructions in
performance-critical operations.
To enable the following instructions: AVX2 AVX_VNNI FMA, in other operations,
rebuild TensorFlow with the appropriate compiler flags.
2025-11-16 22:01:52.400823: I tensorflow/core/util/port.cc:153] oneDNN custom
operations are on. You may see slightly different numerical results due to
floating-point round-off errors from different computation orders. To turn them
off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.
2025-11-16 22:01:52.401004: I external/local_xla/xla/tsl/cuda/cudart_stub.cc:31]
Could not find cuda drivers on your machine, GPU will not be used.
```

This code imports several libraries for machine learning and evaluation. It uses Counter to count data occurrences. sklearn provides tools for classification models like KNeighbors and Random Forest, as well as metrics and cross-validation methods such as GridSearchCV, StratifiedKFold, and train_test_split. Metrics like confusion matrix, ROC AUC, and Brier score are also imported for evaluating models.

It also includes Keras for building deep learning models, like Sequential, Dense, and LSTM layers. Lastly, SMOTE from imblearn is imported to handle class imbalance by oversampling.

[3]:
```
# 1. Loading Data And Preprocessing

# For quick tests, you can switch this to 'waterQuality1_sample.csv'
diab = pd.read_csv('waterQuality1.csv')
print("\nLoaded dataset with shape:", diab.shape)
print("\nDataset info:")
diab.info()

feature_cols = [
    'aluminium', 'ammonia', 'arsenic', 'barium', 'cadmium',
    'chloramine', 'chromium', 'copper', 'flouride', 'bacteria',
    'viruses', 'lead', 'nitrates', 'nitrites', 'mercury',
    'perchlorate', 'radium', 'selenium', 'silver', 'uranium'
]
```

```
print("\nConverting feature columns to numeric...")
diab[feature_cols] = diab[feature_cols].apply(pd.to_numeric, errors='coerce')

print("Converting 'is_safe' to numeric and dropping invalid rows...")
diab['is_safe'] = pd.to_numeric(diab['is_safe'], errors='coerce')
before_drop = diab.shape[0]
diab = diab.dropna(subset=['is_safe'])
after_drop = diab.shape[0]
print(f"Dropped {before_drop - after_drop} rows with invalid 'is_safe' values.")
diab['is_safe'] = diab['is_safe'].astype(int)

def impute_missing_values(dataframe):
    print("\nImputing zeros/NaNs in feature columns with median values...")
    for column in feature_cols:
        zero_count = (dataframe[column] == 0).sum()
        nan_count = dataframe[column].isna().sum()
        if zero_count > 0 or nan_count > 0:
            print(f" - Column '{column}': {zero_count} zeros, {nan_count} NaNs␣
 ↪before imputation")
        dataframe.loc[dataframe[column] == 0, column] = np.nan
        dataframe[column].fillna(dataframe[column].median(), inplace=True)
    return dataframe

diab = impute_missing_values(diab)
print("Finished preprocessing. New dataset shape:", diab.shape)
print("\nPreview of preprocessed data:")
print(diab.head())
```

```
Loaded dataset with shape: (7999, 21)

Dataset info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7999 entries, 0 to 7998
Data columns (total 21 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   aluminium    7999 non-null   float64
 1   ammonia      7999 non-null   object
 2   arsenic      7999 non-null   float64
 3   barium       7999 non-null   float64
 4   cadmium      7999 non-null   float64
 5   chloramine   7999 non-null   float64
 6   chromium     7999 non-null   float64
 7   copper       7999 non-null   float64
 8   flouride     7999 non-null   float64
 9   bacteria     7999 non-null   float64
```

```
 10  viruses      7999 non-null   float64
 11  lead         7999 non-null   float64
 12  nitrates     7999 non-null   float64
 13  nitrites     7999 non-null   float64
 14  mercury      7999 non-null   float64
 15  perchlorate  7999 non-null   float64
 16  radium       7999 non-null   float64
 17  selenium     7999 non-null   float64
 18  silver       7999 non-null   float64
 19  uranium      7999 non-null   float64
 20  is_safe      7999 non-null   object
dtypes: float64(19), object(2)
memory usage: 1.3+ MB

Converting feature columns to numeric…
Converting 'is_safe' to numeric and dropping invalid rows…
Dropped 3 rows with invalid 'is_safe' values.

Imputing zeros/NaNs in feature columns with median values…
 - Column 'aluminium': 251 zeros, 0 NaNs before imputation
 - Column 'ammonia': 6 zeros, 0 NaNs before imputation
 - Column 'arsenic': 187 zeros, 0 NaNs before imputation
 - Column 'barium': 14 zeros, 0 NaNs before imputation
 - Column 'cadmium': 236 zeros, 0 NaNs before imputation
 - Column 'chloramine': 102 zeros, 0 NaNs before imputation
 - Column 'chromium': 208 zeros, 0 NaNs before imputation
 - Column 'copper': 94 zeros, 0 NaNs before imputation
 - Column 'flouride': 25 zeros, 0 NaNs before imputation
 - Column 'bacteria': 2793 zeros, 0 NaNs before imputation
 - Column 'viruses': 1303 zeros, 0 NaNs before imputation
 - Column 'lead': 1 zeros, 0 NaNs before imputation
 - Column 'nitrates': 1 zeros, 0 NaNs before imputation
 - Column 'nitrites': 16 zeros, 0 NaNs before imputation
 - Column 'mercury': 371 zeros, 0 NaNs before imputation
 - Column 'perchlorate': 7 zeros, 0 NaNs before imputation
 - Column 'radium': 18 zeros, 0 NaNs before imputation
 - Column 'selenium': 403 zeros, 0 NaNs before imputation
 - Column 'silver': 246 zeros, 0 NaNs before imputation
 - Column 'uranium': 594 zeros, 0 NaNs before imputation
Finished preprocessing. New dataset shape: (7996, 21)

Preview of preprocessed data:
   aluminium  ammonia  arsenic  barium  cadmium  chloramine  chromium  copper  \
0       1.65     9.08     0.04    2.85    0.007        0.35      0.83    0.17
1       2.32    21.16     0.01    3.31    0.002        5.28      0.68    0.66
2       1.01    14.02     0.04    0.58    0.008        4.24      0.53    0.02
3       1.36    11.33     0.04    2.96    0.001        7.23      0.03    1.66
4       0.92    24.33     0.03    0.20    0.006        2.67      0.69    0.57
```

```
   flouride  bacteria  …   lead  nitrates  nitrites  mercury  perchlorate  \
0      0.05      0.20  …  0.054     16.08      1.13    0.007        37.75
1      0.90      0.65  …  0.100      2.01      1.93    0.003        32.26
2      0.99      0.05  …  0.078     14.16      1.11    0.006        50.28
3      1.08      0.71  …  0.016      1.41      1.29    0.004         9.12
4      0.61      0.13  …  0.117      6.74      1.11    0.003        16.90

   radium  selenium  silver  uranium  is_safe
0    6.78      0.08    0.34     0.02        1
1    3.21      0.08    0.27     0.05        1
2    7.07      0.07    0.44     0.01        0
3    1.72      0.02    0.45     0.05        1
4    2.41      0.02    0.06     0.02        1

[5 rows x 21 columns]
```

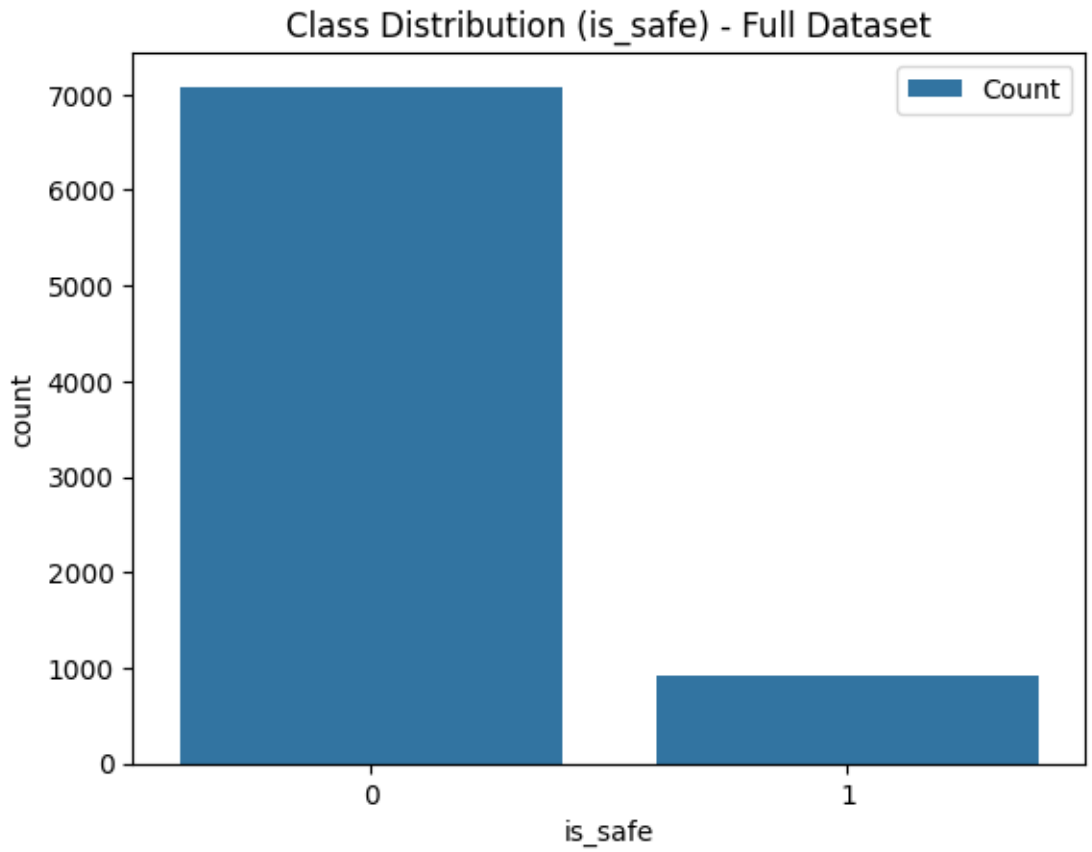
This code loads and preprocesses a water quality dataset. It reads the data from a CSV file, then prints the dataset's shape and info. It defines a list of feature columns and converts them to numbers, handling errors by coercing.

The 'is_safe' column is also converted to numbers, and any rows with invalid data are removed. The number of dropped rows is shown. The 'is_safe' column is then changed to integers. A function replaces zeros and NaNs in the features with the median value of each column. After processing, it prints the new dataset shape and shows a preview of the cleaned data.

[4]:
```
# 2. Separating Dataset into Features and Output Label

features = diab.iloc[:, :-1]
labels = diab.iloc[:, -1]

print("Features shape:", features.shape)
print("Labels shape:", labels.shape)

print("\nClass distribution (is_safe):")
print(labels.value_counts())
positive_outcomes, negative_outcomes = labels.value_counts()
total_samples = labels.count()
print('\n----------Checking for Data Imbalance------------')
print('Number of Positive Outcomes: ', positive_outcomes)
print('Percentage of Positive Outcomes: {}%'.format(round((positive_outcomes /␣
 ↪total_samples) * 100, 2)))
print('Number of Negative Outcomes : ', negative_outcomes)
print('Percentage of Negative Outcomes: {}%'.format(round((negative_outcomes /␣
 ↪total_samples) * 100, 2)))
print('-------------------------------------------------\n')

sns.countplot(x=labels, label="Count")
```

```
plt.title("Class Distribution (is_safe) - Full Dataset")
plt.savefig("class_distribution_full.png", bbox_inches="tight")
plt.show()
```

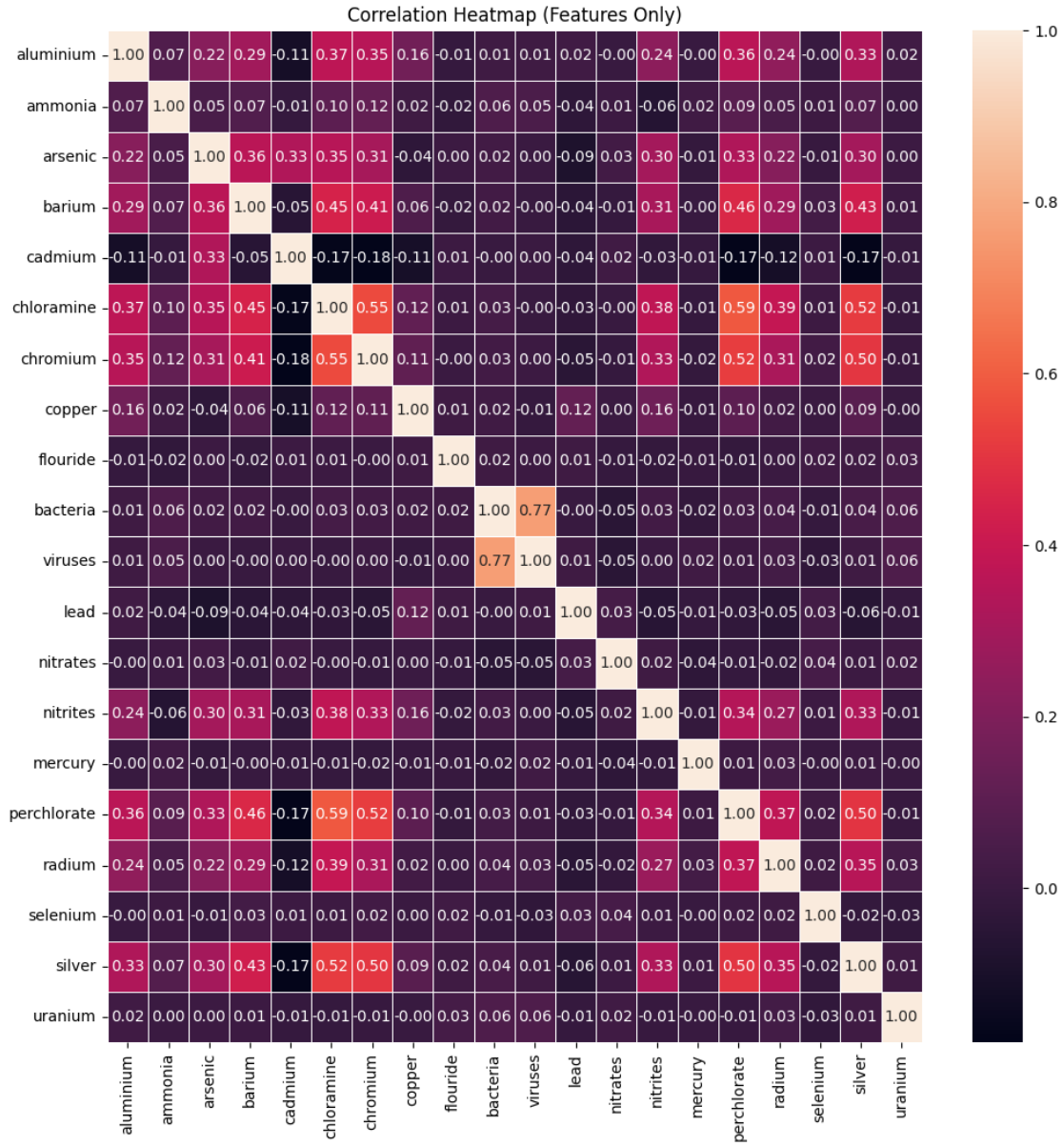
```
Features shape: (7996, 20)
Labels shape: (7996,)

Class distribution (is_safe):
is_safe
0    7084
1     912
Name: count, dtype: int64

----------Checking for Data Imbalance------------
Number of Positive Outcomes:  7084
Percentage of Positive Outcomes: 88.59%
Number of Negative Outcomes :  912
Percentage of Negative Outcomes: 11.41%
-------------------------------------------------
```



This code splits the dataset into features and labels. It uses iloc to select all columns except the last for features, and the last column for labels. It then prints their shapes.

Next, it shows how many 'is_safe' labels are positive or negative. It calculates and displays the percentage of each to check for imbalance. A seaborn count plot is created to show the class distribution visually, and saved as "class_distribution_full.png".

[5]:
```
# 3. Checking Correlation Between Attributes

correlation_matrix = features.corr()
fig, axis = plt.subplots(figsize=(12, 12))
sns.heatmap(correlation_matrix, annot=True, linewidths=.5, fmt='.2f', ax=axis)
plt.title("Correlation Heatmap (Features Only)")
plt.savefig("correlation_heatmap.png", bbox_inches="tight")
plt.show()


full_corr = diab.corr()
corr_with_target = full_corr['is_safe'].drop('is_safe')
max_corr_feat = corr_with_target.abs().idxmax()
print("\nCorrelation of each feature with 'is_safe':")
print(corr_with_target.sort_values(ascending=False))
print(f"\nHighest absolute correlation with 'is_safe': {max_corr_feat} "
      f"(corr = {corr_with_target[max_corr_feat]:.4f})")
```



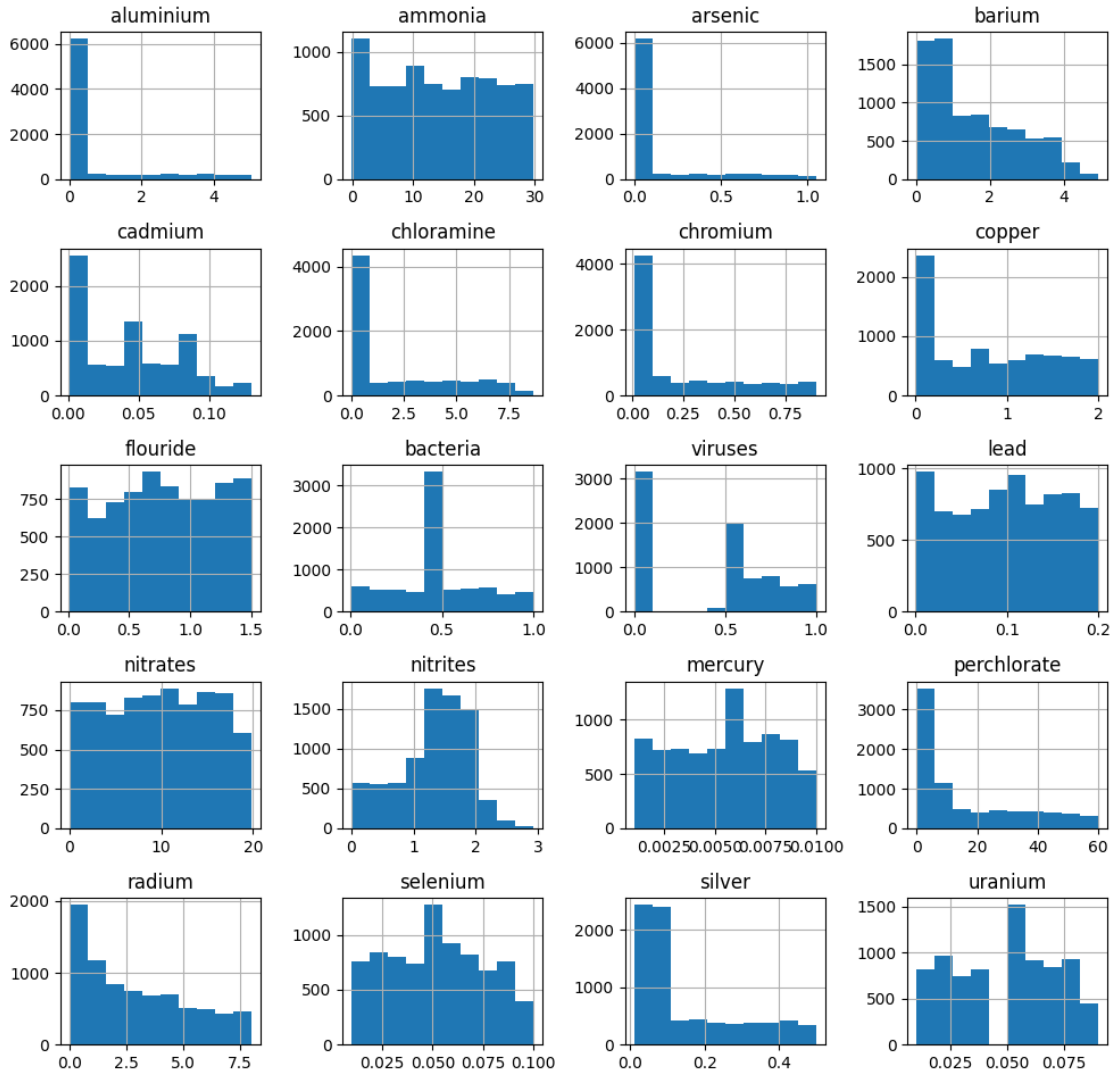

```
Correlation of each feature with 'is_safe':
aluminium      0.333740
chloramine     0.186434
chromium       0.182355
silver         0.102017
barium         0.090753
perchlorate    0.075791
radium         0.065655
nitrites       0.048392
copper         0.028511
```

```
flouride       0.005288
lead          -0.009360
bacteria      -0.018284
ammonia       -0.022743
selenium      -0.024732
mercury       -0.030667
viruses       -0.045848
nitrates      -0.071490
uranium       -0.077183
arsenic       -0.124948
cadmium       -0.271881
Name: is_safe, dtype: float64

Highest absolute correlation with 'is_safe': aluminium (corr = 0.3337)
```

This code checks how the features in the dataset are related. It calculates a correlation matrix for the feature columns and creates a heatmap to visualize these relationships. The heatmap is saved as "correlation_heatmap.png".

Then, it finds the correlation of each feature with the target label 'is_safe', excluding 'is_safe' itself. It identifies and prints the feature that has the strongest absolute correlation with 'is_safe', along with the correlation values for all features in order from highest to lowest.

[6]:
```python
# 4. Visualizing Distributions (Histograms) & Symmetry

features.hist(figsize=(10, 10))
plt.suptitle("Histograms of Features", y=1.02)
plt.tight_layout()
plt.savefig("feature_histograms.png", bbox_inches="tight")
plt.show()

print("Histograms plotted for all feature columns.")

skewness = features.skew()
print("\nSkewness of each feature:")
print(skewness.sort_values())

sym_threshold = 0.5
approx_symmetric = skewness[skewness.abs() < sym_threshold].index.tolist()
print(f"\nFeatures with approximate symmetry (|skew| < {sym_threshold}):")
print(approx_symmetric if approx_symmetric else "None")
```



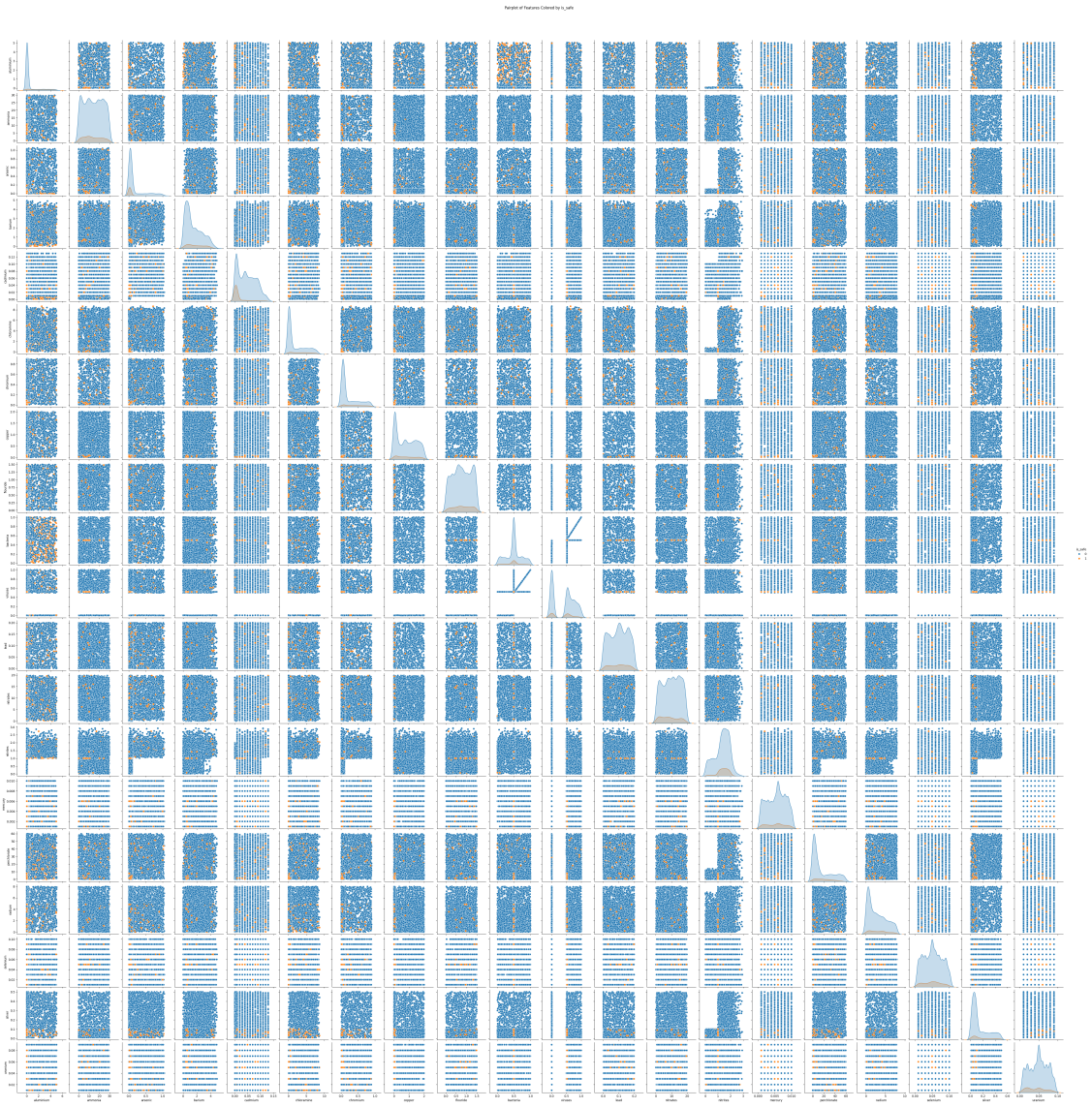

```
Histograms plotted for all feature columns.

Skewness of each feature:
nitrites      -0.495900
mercury       -0.092567
lead          -0.060667
viruses       -0.052418
nitrates      -0.042061
flouride      -0.039333
uranium       -0.016994
bacteria      -0.014163
ammonia        0.026424
```

```
selenium       0.062251
copper         0.241822
cadmium        0.462695
radium         0.548674
barium         0.662255
chloramine     0.889899
perchlorate    0.937986
chromium       1.037485
silver         1.048531
arsenic        1.990500
aluminium      2.014929
dtype: float64

Features with approximate symmetry (|skew| < 0.5):
['ammonia', 'cadmium', 'copper', 'flouride', 'bacteria', 'viruses', 'lead',
'nitrates', 'nitrites', 'mercury', 'selenium', 'uranium']
```

This code creates histograms for each feature to show their distributions. The histograms are arranged in a 10x10 grid and saved as "feature_histograms.png". A message is printed to confirm the plots are created.

Next, it calculates the skewness of each feature to check how symmetric their distributions are. The skewness values are printed from smallest to largest. Features with skewness close to zero (less than 0.5 in absolute value) are considered roughly symmetric, and these features are identified and printed.

[7]:
```
# 5. Pairwise Relationships (Pairplot)

sns.pairplot(diab, hue='is_safe')
plt.suptitle("Pairplot of Features Colored by is_safe", y=1.02)
plt.savefig("pairplot_is_safe.png", bbox_inches="tight")
plt.show()

print("Pairplot generated.")
```



```
Pairplot generated.
```

This code creates a pairplot to show the relationships between all features in the dataset. The points are colored based on the 'is_safe' label. The plot is saved as "pairplot_is_safe.png". A message is printed to confirm the pairplot has been created.

[8]:
```
# 6. Train-Test Split and Normalization (70/30 Split)

features_train_all, features_test_all, labels_train_all, labels_test_all =␣
  ↪train_test_split(
    features, labels, test_size=0.3, random_state=42, stratify=labels
)
```

```
for dataset in [features_train_all, features_test_all, labels_train_all,␣
 ↪labels_test_all]:
    dataset.reset_index(drop=True, inplace=True)

print("Training set shape:", features_train_all.shape)
print("Test set shape:", features_test_all.shape)

features_train_all_std = (features_train_all - features_train_all.mean()) /␣
 ↪features_train_all.std()
features_test_all_std = (features_test_all - features_test_all.mean()) /␣
 ↪features_test_all.std()

print("\nTraining data (standardized) summary:")
print(features_train_all_std.describe())

n_features = features_train_all_std.shape[1]
print("Number of features (for LSTM):", n_features)
```

```
Training set shape: (5597, 20)
Test set shape: (2399, 20)

Training data (standardized) summary:
          aluminium       ammonia       arsenic        barium       cadmium  \
count  5.597000e+03  5.597000e+03  5.597000e+03  5.597000e+03  5.597000e+03
mean  -1.396457e-17  1.491670e-16  7.045761e-17 -4.633698e-17 -6.474483e-17
std    1.000000e+00  1.000000e+00  1.000000e+00  1.000000e+00  1.000000e+00
min   -5.220567e-01 -1.626751e+00 -6.415196e-01 -1.283456e+00 -1.221047e+00
25%   -4.986138e-01 -8.626112e-01 -5.250050e-01 -8.307739e-01 -9.949998e-01
50%   -4.673567e-01 -1.406668e-02 -4.446501e-01 -3.122473e-01 -1.190666e-01
75%   -3.110708e-01  8.873712e-01 -2.437630e-01  7.412669e-01  7.286107e-01
max    3.416347e+00  1.740417e+00  3.573093e+00  2.774220e+00  2.423965e+00

         chloramine      chromium        copper      flouride      bacteria  \
count  5.597000e+03  5.597000e+03  5.597000e+03  5.597000e+03  5.597000e+03
mean  -6.760122e-17  2.126423e-17 -6.601433e-17  3.618093e-16  1.142556e-17
std    1.000000e+00  1.000000e+00  1.000000e+00  1.000000e+00  1.000000e+00
min   -8.442890e-01 -8.909774e-01 -1.252761e+00 -1.762467e+00 -2.091531e+00
25%   -8.090765e-01 -7.429517e-01 -1.114053e+00 -8.427510e-01 -4.898023e-01
50%   -6.291014e-01 -5.949260e-01 -9.686630e-02 -1.500680e-02  2.967725e-02
75%    7.793996e-01  7.373052e-01  8.740850e-01  8.817161e-01  4.625769e-01
max    2.493076e+00  2.402594e+00  1.814212e+00  1.663475e+00  2.194175e+00

           viruses          lead      nitrates      nitrites       mercury  \
count  5597.000000  5.597000e+03  5.597000e+03  5.597000e+03  5.597000e+03
mean      0.000000 -1.777309e-17  2.278764e-16  1.015605e-17 -1.485322e-16
std       1.000000  1.000000e+00  1.000000e+00  1.000000e+00  1.000000e+00
min      -1.163468 -1.696924e+00 -1.778221e+00 -2.327137e+00 -1.628551e+00
```

```
25%      -1.149252 -8.890785e-01 -8.509969e-01 -5.722683e-01 -9.000794e-01
50%       0.312161  3.908419e-02  1.465387e-02  1.472277e-01  1.926278e-01
75%       0.823940  8.984940e-01  8.640058e-01  7.438830e-01  9.210992e-01
max       1.676905  1.723528e+00  1.811151e+00  2.779530e+00  1.649571e+00

        perchlorate        radium      selenium        silver       uranium
count  5.597000e+03  5.597000e+03  5.597000e+03  5.597000e+03  5.597000e+03
mean  -5.458878e-17  8.505693e-17  1.142556e-16 -2.012168e-16 -2.183551e-16
std    1.000000e+00  1.000000e+00  1.000000e+00  1.000000e+00  1.000000e+00
min   -9.359252e-01 -1.264000e+00 -1.611566e+00 -9.912686e-01 -1.606983e+00
25%   -8.134339e-01 -9.102931e-01 -8.523122e-01 -7.068548e-01 -7.677685e-01
50%   -4.941626e-01 -2.201339e-01 -9.305844e-02 -4.224411e-01  7.144648e-02
75%    7.466287e-01  7.547160e-01  6.661953e-01  6.441105e-01  9.106615e-01
max    2.466611e+00  2.178169e+00  1.805076e+00  2.492800e+00  1.749876e+00
Number of features (for LSTM): 20
```

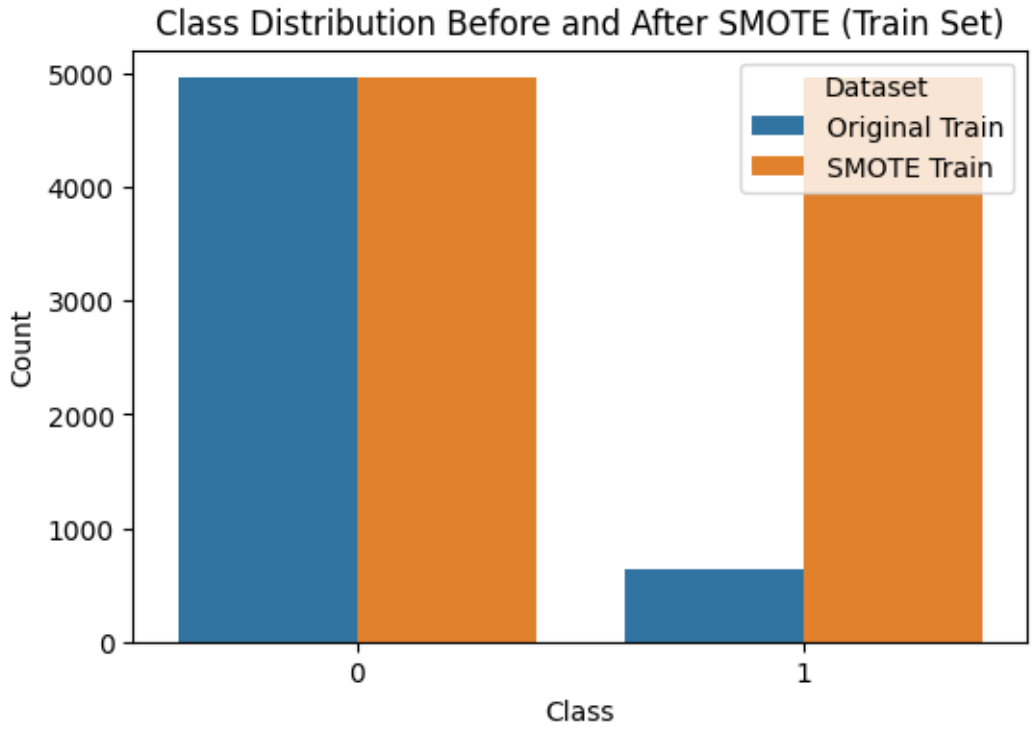
This code splits the dataset into training and testing sets, with 70% for training and 30% for testing. It uses train_test_split with stratification to keep class proportions the same. The indices of both sets are reset for consistency.

Then, it standardizes the features by subtracting the mean and dividing by the standard deviation for both training and test data. It prints a summary of the standardized training data and shows the number of features used for the LSTM model.

[9]:
```python
# 7. Visualizing Effect of SMOTE on Training Data

smote = SMOTE(random_state=42)

print("\nOriginal training label distribution:")
orig_counts = Counter(labels_train_all)
print(orig_counts)

# Apply SMOTE to the training data only
features_train_bal_all, labels_train_bal_all = smote.
 ↪fit_resample(features_train_all_std, labels_train_all)

print("\nBalanced training label distribution after SMOTE:")
bal_counts = Counter(labels_train_bal_all)
print(bal_counts)

# Print class distribution before and after SMOTE
print("\nClass distribution before and after SMOTE:")
print(f"Original training set class distribution: {orig_counts}")
print(f"Balanced training set class distribution after SMOTE: {bal_counts}")

# Visualize the effect of SMOTE
smote_df = pd.DataFrame({
    'Class': ['0', '1', '0', '1'],
```

```
    'Count': [
        orig_counts.get(0, 0), orig_counts.get(1, 0),
        bal_counts.get(0, 0), bal_counts.get(1, 0)
    ],
    'Dataset': ['Original Train', 'Original Train',
                'SMOTE Train', 'SMOTE Train']
})

plt.figure(figsize=(6, 4))
sns.barplot(data=smote_df, x='Class', y='Count', hue='Dataset')
plt.title("Class Distribution Before and After SMOTE (Train Set)")
plt.savefig("smote_class_balance.png", bbox_inches="tight")
plt.show()
```

```
Original training label distribution:
Counter({0: 4959, 1: 638})

Balanced training label distribution after SMOTE:
Counter({0: 4959, 1: 4959})

Class distribution before and after SMOTE:
Original training set class distribution: Counter({0: 4959, 1: 638})
Balanced training set class distribution after SMOTE: Counter({0: 4959, 1:
4959})
```



This code shows the class distribution of the training labels before applying SMOTE, using print to display the counts. It then applies SMOTE to balance the classes by oversampling the minority class.

After SMOTE, it prints the new class distribution and compares it to the original. A bar plot is created to visually compare the class balance before and after SMOTE, and it is saved as "smote_class_balance.png".

[10]:
```
# 8. Defining Metric Calculation and Model Evaluation Functions

def calc_metrics(confusion_matrix_):
    TP, FN = confusion_matrix_[0][0], confusion_matrix_[0][1]
    FP, TN = confusion_matrix_[1][0], confusion_matrix_[1][1]

    TPR = TP / (TP + FN) if TP + FN > 0 else 0
    TNR = TN / (TN + FP) if TN + FP > 0 else 0
    FPR = FP / (TN + FP) if TN + FP > 0 else 0
    FNR = FN / (TP + FN) if TP + FN > 0 else 0
    Precision = TP / (TP + FP) if TP + FP > 0 else 0
    F1_measure = 2 * TP / (2 * TP + FP + FN) if (2 * TP + FP + FN) > 0 else 0
    Accuracy = (TP + TN) / (TP + FP + FN + TN) if (TP + FP + FN + TN) > 0 else 0
    Error_rate = 1 - Accuracy
    BACC = (TPR + TNR) / 2
    TSS = TPR - FPR
    denom_hss = ((TP + FN) * (FN + TN) + (TP + FP) * (FP + TN))
    HSS = 2 * (TP * TN - FP * FN) / denom_hss if denom_hss != 0 else 0

    metrics = [
        TP, TN, FP, FN, TPR, TNR, FPR, FNR,
        Precision, F1_measure, Accuracy, Error_rate,
        BACC, TSS, HSS
    ]
    return metrics

def get_metrics(model, features_train, features_test, labels_train,␣
 ↪labels_test, LSTM_flag):

    metrics = []

    if LSTM_flag == 1:
        Xtrain, Xtest, ytrain, ytest = map(
            np.array, [features_train, features_test, labels_train, labels_test]
        )

        shape = Xtrain.shape
```

```
        Xtrain_reshaped = Xtrain.reshape(len(Xtrain), shape[1], 1)
        Xtest_reshaped = Xtest.reshape(len(Xtest), shape[1], 1)

        model.fit(
            Xtrain_reshaped, ytrain,
            epochs=50,
            validation_data=(Xtest_reshaped, ytest),
            verbose=0
        )

        lstm_scores = model.evaluate(Xtest_reshaped, ytest, verbose=0)
        predict_prob = model.predict(Xtest_reshaped, verbose=0)
        pred_labels = (predict_prob > 0.5).astype(int)

        matrix = confusion_matrix(ytest, pred_labels, labels=[1, 0])
        brier = brier_score_loss(ytest, predict_prob)
        roc_auc = roc_auc_score(ytest, predict_prob)

        metrics.extend(calc_metrics(matrix))
        metrics.extend([brier, roc_auc, lstm_scores[1]])

    else:
        model.fit(features_train, labels_train)
        predicted = model.predict(features_test)
        matrix = confusion_matrix(labels_test, predicted, labels=[1, 0])
        proba = model.predict_proba(features_test)[:, 1]
        brier = brier_score_loss(labels_test, proba)
        roc_auc = roc_auc_score(labels_test, proba)

        metrics.extend(calc_metrics(matrix))
        metrics.extend([brier, roc_auc, model.score(features_test,␣
↪labels_test)])

    return metrics
```

This code defines two functions for evaluating a machine learning model. The first function, calc_metrics, takes a confusion matrix and computes various metrics like TPR, TNR, FPR, FNR, Precision, F1-score, Accuracy, Error rate, BACC, TSS, and HSS. These help measure how well the model performs in binary classification.

The second function, get_metrics, evaluates a given model on training and test data. If using an LSTM model (indicated by LSTM_flag), it reshapes the data, trains the model for 50 epochs, and calculates the confusion matrix, Brier score, ROC AUC, and accuracy. For other models, it fits the model, makes predictions, and computes the same metrics. It returns a list of all these metrics for detailed performance analysis.

[11]:
```
# 9. Hyperparameter Tuning (KNN & Random Forest)
```

```
knn_parameters = {"n_neighbors": list(range(1, 16))}
knn_model = KNeighborsClassifier(n_jobs=-1)
knn_cv = GridSearchCV(knn_model, knn_parameters, cv=10, n_jobs=-1)
knn_cv.fit(features_train_all_std, labels_train_all)
best_n_neighbors = knn_cv.best_params_["n_neighbors"]
print("Best KNN n_neighbors:", best_n_neighbors)

param_grid_rf = {
    "n_estimators": list(range(10, 101, 10)),
    "min_samples_split": [2, 4, 6, 8, 10]
}
rf_classifier = RandomForestClassifier(n_jobs=-1)
grid_search_rf = GridSearchCV(
    estimator=rf_classifier,
    param_grid=param_grid_rf,
    cv=10,
    n_jobs=-1
)
grid_search_rf.fit(features_train_all_std, labels_train_all)
best_rf_params = grid_search_rf.best_params_
min_samples_split = best_rf_params["min_samples_split"]
n_estimators = best_rf_params["n_estimators"]
print("Best RF params:", best_rf_params)
```

```
Best KNN n_neighbors: 13
Best RF params: {'min_samples_split': 4, 'n_estimators': 90}
```

This code performs hyperparameter tuning for K-Nearest Neighbors (KNN) and Random Forest models using GridSearchCV.

For KNN, it sets up a grid of n_neighbors values from 1 to 15. The model is trained with 10-fold cross-validation, and the optimal number of neighbors is identified via best_params_. The best value is then printed.

For Random Forest, it creates a parameter grid to tune n_estimators from 10 to 100 in steps of 10, and min_samples_split from 2 to 10. The model is trained with 10-fold cross-validation to find the best parameter combination. The optimal parameters, including the number of estimators and minimum samples to split, are printed.

[12]:
```
# 10. 10-fold Stratified Cross-Validation with SMOTE

cv_stratified = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)

metric_columns = [
    'TP', 'TN', 'FP', 'FN', 'TPR', 'TNR', 'FPR', 'FNR',
    'Precision', 'F1_measure', 'Accuracy', 'Error_rate',
    'BACC', 'TSS', 'HSS', 'Brier_score', 'AUC', 'Acc_by_package_fn'
]
```

```
knn_metrics_list = []
rf_metrics_list = []
lstm_metrics_list = []

for iter_num, (train_index, test_index) in enumerate(
    cv_stratified.split(features_train_all_std, labels_train_all), start=1
):
    print(f"\n--- Cross-Validation Iteration {iter_num} ---")

    knn_model = KNeighborsClassifier(n_neighbors=best_n_neighbors, n_jobs=-1)
    rf_model = RandomForestClassifier(
        min_samples_split=min_samples_split,
        n_estimators=n_estimators,
        n_jobs=-1
    )

    lstm_model = Sequential()
    lstm_model.add(
        LSTM(
            64,
            activation='relu',
            input_shape=(n_features, 1),
            return_sequences=False
        )
    )
    lstm_model.add(Dense(1, activation='sigmoid'))
    lstm_model.compile(loss='binary_crossentropy', optimizer='adam',␣
 ↪metrics=['accuracy'])

    features_train = features_train_all_std.iloc[train_index, :]
    features_test = features_train_all_std.iloc[test_index, :]
    labels_train = labels_train_all[train_index]
    labels_test = labels_train_all[test_index]

    features_train_bal, labels_train_bal = smote.fit_resample(features_train,␣
 ↪labels_train)

    knn_metrics = get_metrics(knn_model, features_train_bal, features_test,␣
 ↪labels_train_bal, labels_test, 0)
    rf_metrics = get_metrics(rf_model, features_train_bal, features_test,␣
 ↪labels_train_bal, labels_test, 0)
    lstm_metrics = get_metrics(lstm_model, features_train_bal, features_test,␣
 ↪labels_train_bal, labels_test, 1)

    knn_metrics_list.append(knn_metrics)
    rf_metrics_list.append(rf_metrics)
    lstm_metrics_list.append(lstm_metrics)
```

```
    iter_df = pd.DataFrame(
        {
            "KNN": knn_metrics,
            "RF": rf_metrics,
            "LSTM": lstm_metrics
        },
        index=metric_columns
    )

    print(f"\nIteration {iter_num}:")
    print(f"----- Metrics for all Algorithms in Iteration {iter_num} -----")
    print(iter_df.round(2).to_string())
```

```
--- Cross-Validation Iteration 1 ---
WARNING: All log messages before absl::InitializeLog() is called are written to
STDERR
E0000 00:00:1763348662.988984 3504613 cuda_executor.cc:1309] INTERNAL: CUDA
Runtime error: Failed call to cudaGetRuntimeVersion: Error loading CUDA
libraries. GPU will not be used.: Error loading CUDA libraries. GPU will not be
used.
W0000 00:00:1763348662.993842 3504613 gpu_device.cc:2342] Cannot dlopen some GPU
libraries. Please make sure the missing libraries mentioned above are installed
properly if you would like to use GPU. Follow the guide at
https://www.tensorflow.org/install/gpu for how to download and setup the
required libraries for your platform.
Skipping registering GPU devices…


Iteration 1:
----- Metrics for all Algorithms in Iteration 1 -----
                      KNN      RF    LSTM
TP                  58.00   52.00   50.00
TN                 362.00  489.00  483.00
FP                 134.00    7.00   13.00
FN                   6.00   12.00   14.00
TPR                  0.91    0.81    0.78
TNR                  0.73    0.99    0.97
FPR                  0.27    0.01    0.03
FNR                  0.09    0.19    0.22
Precision            0.30    0.88    0.79
F1_measure           0.45    0.85    0.79
Accuracy             0.75    0.97    0.95
Error_rate           0.25    0.03    0.05
BACC                 0.82    0.90    0.88
TSS                  0.64    0.80    0.76
HSS                  0.34    0.83    0.76
```

```
Brier_score          0.17    0.04    0.04
AUC                  0.90    0.98    0.95
Acc_by_package_fn    0.75    0.97    0.95

--- Cross-Validation Iteration 2 ---

Iteration 2:
----- Metrics for all Algorithms in Iteration 2 -----
                       KNN      RF    LSTM
TP                   61.00   50.00   52.00
TN                  366.00  477.00  463.00
FP                  130.00   19.00   33.00
FN                    3.00   14.00   12.00
TPR                   0.95    0.78    0.81
TNR                   0.74    0.96    0.93
FPR                   0.26    0.04    0.07
FNR                   0.05    0.22    0.19
Precision             0.32    0.72    0.61
F1_measure            0.48    0.75    0.70
Accuracy              0.76    0.94    0.92
Error_rate            0.24    0.06    0.08
BACC                  0.85    0.87    0.87
TSS                   0.69    0.74    0.75
HSS                   0.37    0.72    0.65
Brier_score           0.16    0.05    0.06
AUC                   0.92    0.96    0.96
Acc_by_package_fn     0.76    0.94    0.92

--- Cross-Validation Iteration 3 ---

Iteration 3:
----- Metrics for all Algorithms in Iteration 3 -----
                       KNN      RF    LSTM
TP                   59.00   50.00   50.00
TN                  352.00  480.00  461.00
FP                  144.00   16.00   35.00
FN                    5.00   14.00   14.00
TPR                   0.92    0.78    0.78
TNR                   0.71    0.97    0.93
FPR                   0.29    0.03    0.07
FNR                   0.08    0.22    0.22
Precision             0.29    0.76    0.59
F1_measure            0.44    0.77    0.67
Accuracy              0.73    0.95    0.91
Error_rate            0.27    0.05    0.09
BACC                  0.82    0.87    0.86
TSS                   0.63    0.75    0.71
HSS                   0.32    0.74    0.62
```

```
Brier_score          0.18    0.04    0.07
AUC                  0.88    0.97    0.94
Acc_by_package_fn    0.73    0.95    0.91

--- Cross-Validation Iteration 4 ---

Iteration 4:
----- Metrics for all Algorithms in Iteration 4 -----
                       KNN      RF    LSTM
TP                   58.00   51.00   51.00
TN                  377.00  482.00  455.00
FP                  119.00   14.00   41.00
FN                    6.00   13.00   13.00
TPR                   0.91    0.80    0.80
TNR                   0.76    0.97    0.92
FPR                   0.24    0.03    0.08
FNR                   0.09    0.20    0.20
Precision             0.33    0.78    0.55
F1_measure            0.48    0.79    0.65
Accuracy              0.78    0.95    0.90
Error_rate            0.22    0.05    0.10
BACC                  0.83    0.88    0.86
TSS                   0.67    0.77    0.71
HSS                   0.38    0.76    0.60
Brier_score           0.16    0.04    0.08
AUC                   0.88    0.97    0.93
Acc_by_package_fn     0.78    0.95    0.90

--- Cross-Validation Iteration 5 ---

Iteration 5:
----- Metrics for all Algorithms in Iteration 5 -----
                       KNN      RF    LSTM
TP                   59.00   51.00   52.00
TN                  352.00  488.00  477.00
FP                  144.00    8.00   19.00
FN                    5.00   13.00   12.00
TPR                   0.92    0.80    0.81
TNR                   0.71    0.98    0.96
FPR                   0.29    0.02    0.04
FNR                   0.08    0.20    0.19
Precision             0.29    0.86    0.73
F1_measure            0.44    0.83    0.77
Accuracy              0.73    0.96    0.94
Error_rate            0.27    0.04    0.06
BACC                  0.82    0.89    0.89
TSS                   0.63    0.78    0.77
HSS                   0.32    0.81    0.74
```

```
Brier_score          0.17    0.03    0.05
AUC                  0.89    0.98    0.96
Acc_by_package_fn    0.73    0.96    0.94

--- Cross-Validation Iteration 6 ---

Iteration 6:
----- Metrics for all Algorithms in Iteration 6 -----
                       KNN      RF    LSTM
TP                   56.00   48.00   45.00
TN                  363.00  481.00  474.00
FP                  133.00   15.00   22.00
FN                    8.00   16.00   19.00
TPR                   0.88    0.75    0.70
TNR                   0.73    0.97    0.96
FPR                   0.27    0.03    0.04
FNR                   0.12    0.25    0.30
Precision             0.30    0.76    0.67
F1_measure            0.44    0.76    0.69
Accuracy              0.75    0.94    0.93
Error_rate            0.25    0.06    0.07
BACC                  0.80    0.86    0.83
TSS                   0.61    0.72    0.66
HSS                   0.33    0.72    0.65
Brier_score           0.18    0.05    0.06
AUC                   0.87    0.96    0.94
Acc_by_package_fn     0.75    0.94    0.93

--- Cross-Validation Iteration 7 ---

Iteration 7:
----- Metrics for all Algorithms in Iteration 7 -----
                       KNN      RF    LSTM
TP                   57.00   49.00   49.00
TN                  356.00  475.00  467.00
FP                  140.00   21.00   29.00
FN                    7.00   15.00   15.00
TPR                   0.89    0.77    0.77
TNR                   0.72    0.96    0.94
FPR                   0.28    0.04    0.06
FNR                   0.11    0.23    0.23
Precision             0.29    0.70    0.63
F1_measure            0.44    0.73    0.69
Accuracy              0.74    0.94    0.92
Error_rate            0.26    0.06    0.08
BACC                  0.80    0.86    0.85
TSS                   0.61    0.72    0.71
HSS                   0.32    0.69    0.65
```

```
Brier_score          0.19    0.05    0.06
AUC                  0.86    0.97    0.95
Acc_by_package_fn    0.74    0.94    0.92

--- Cross-Validation Iteration 8 ---

Iteration 8:
----- Metrics for all Algorithms in Iteration 8 -----
                       KNN      RF    LSTM
TP                   60.00   46.00   46.00
TN                  349.00  480.00  478.00
FP                  147.00   16.00   18.00
FN                    3.00   17.00   17.00
TPR                   0.95    0.73    0.73
TNR                   0.70    0.97    0.96
FPR                   0.30    0.03    0.04
FNR                   0.05    0.27    0.27
Precision             0.29    0.74    0.72
F1_measure            0.44    0.74    0.72
Accuracy              0.73    0.94    0.94
Error_rate            0.27    0.06    0.06
BACC                  0.83    0.85    0.85
TSS                   0.66    0.70    0.69
HSS                   0.33    0.70    0.69
Brier_score           0.18    0.04    0.05
AUC                   0.89    0.98    0.96
Acc_by_package_fn     0.73    0.94    0.94

--- Cross-Validation Iteration 9 ---

Iteration 9:
----- Metrics for all Algorithms in Iteration 9 -----
                       KNN      RF    LSTM
TP                   59.00   46.00   46.00
TN                  363.00  480.00  467.00
FP                  133.00   16.00   29.00
FN                    4.00   17.00   17.00
TPR                   0.94    0.73    0.73
TNR                   0.73    0.97    0.94
FPR                   0.27    0.03    0.06
FNR                   0.06    0.27    0.27
Precision             0.31    0.74    0.61
F1_measure            0.46    0.74    0.67
Accuracy              0.75    0.94    0.92
Error_rate            0.25    0.06    0.08
BACC                  0.83    0.85    0.84
TSS                   0.67    0.70    0.67
HSS                   0.35    0.70    0.62
```

```
Brier_score          0.17    0.05    0.06
AUC                  0.89    0.96    0.94
Acc_by_package_fn    0.75    0.94    0.92

--- Cross-Validation Iteration 10 ---

Iteration 10:
----- Metrics for all Algorithms in Iteration 10 -----
                       KNN      RF    LSTM
TP                   60.00   57.00   48.00
TN                  348.00  483.00  460.00
FP                  147.00   12.00   35.00
FN                    4.00    7.00   16.00
TPR                   0.94    0.89    0.75
TNR                   0.70    0.98    0.93
FPR                   0.30    0.02    0.07
FNR                   0.06    0.11    0.25
Precision             0.29    0.83    0.58
F1_measure            0.44    0.86    0.65
Accuracy              0.73    0.97    0.91
Error_rate            0.27    0.03    0.09
BACC                  0.82    0.93    0.84
TSS                   0.64    0.87    0.68
HSS                   0.32    0.84    0.60
Brier_score           0.18    0.04    0.07
AUC                   0.90    0.98    0.93
Acc_by_package_fn     0.73    0.97    0.91
```

This code performs 10-fold stratified cross-validation with SMOTE to balance the training data. It uses StratifiedKFold to ensure each fold maintains the class distribution. In each fold, it splits the data into training and testing sets based on the generated indices.

SMOTE is applied to the training data to address class imbalance. Then, it trains and evaluates three models KNN, Random Forest, and LSTM using the get_metrics function, storing their performance metrics in separate lists.

After each fold, a DataFrame is created to display the metrics for all models, and the results are printed. These metrics include TPR, FPR, F1-score, AUC, and others, providing a comprehensive assessment of model performance across all folds.

[13]:
```python
# 11. Average Performance & Per-Algorithm Iteration Tables

metric_index_df = [
    'iter1', 'iter2', 'iter3', 'iter4', 'iter5',
    'iter6', 'iter7', 'iter8', 'iter9', 'iter10'
]

knn_metrics_df = pd.DataFrame(knn_metrics_list, columns=metric_columns,␣
 ↪index=metric_index_df)
```

```
rf_metrics_df = pd.DataFrame(rf_metrics_list, columns=metric_columns,␣
 ↪index=metric_index_df)
lstm_metrics_df = pd.DataFrame(lstm_metrics_list, columns=metric_columns,␣
 ↪index=metric_index_df)

print("\nMetrics for Algorithm 1 (KNN):")
print(knn_metrics_df.T.round(2).to_string())

print("\nMetrics for Algorithm 2 (RF):")
print(rf_metrics_df.T.round(2).to_string())

print("\nMetrics for Algorithm 3 (LSTM):")
print(lstm_metrics_df.T.round(2).to_string())

knn_avg_df = knn_metrics_df.mean()
rf_avg_df = rf_metrics_df.mean()
lstm_avg_df = lstm_metrics_df.mean()

avg_performance_df = pd.DataFrame(
    {"KNN": knn_avg_df, "RF": rf_avg_df, "LSTM": lstm_avg_df},
    index=metric_columns
)

print("\nAverage performance over 10 folds (rounded):")
print(avg_performance_df.round(3))
```

```
Metrics for Algorithm 1 (KNN):
                   iter1   iter2   iter3   iter4   iter5   iter6   iter7
iter8   iter9  iter10
TP                 58.00   61.00   59.00   58.00   59.00   56.00   57.00
60.00   59.00   60.00
TN                362.00  366.00  352.00  377.00  352.00  363.00  356.00
349.00  363.00  348.00
FP                134.00  130.00  144.00  119.00  144.00  133.00  140.00
147.00  133.00  147.00
FN                  6.00    3.00    5.00    6.00    5.00    8.00    7.00
3.00    4.00    4.00
TPR                 0.91    0.95    0.92    0.91    0.92    0.88    0.89
0.95    0.94    0.94
TNR                 0.73    0.74    0.71    0.76    0.71    0.73    0.72
0.70    0.73    0.70
FPR                 0.27    0.26    0.29    0.24    0.29    0.27    0.28
0.30    0.27    0.30
FNR                 0.09    0.05    0.08    0.09    0.08    0.12    0.11
0.05    0.06    0.06
Precision           0.30    0.32    0.29    0.33    0.29    0.30    0.29
```

```
0.29    0.31    0.29
F1_measure           0.45    0.48    0.44    0.48    0.44    0.44    0.44
0.44    0.46    0.44
Accuracy             0.75    0.76    0.73    0.78    0.73    0.75    0.74
0.73    0.75    0.73
Error_rate           0.25    0.24    0.27    0.22    0.27    0.25    0.26
0.27    0.25    0.27
BACC                 0.82    0.85    0.82    0.83    0.82    0.80    0.80
0.83    0.83    0.82
TSS                  0.64    0.69    0.63    0.67    0.63    0.61    0.61
0.66    0.67    0.64
HSS                  0.34    0.37    0.32    0.38    0.32    0.33    0.32
0.33    0.35    0.32
Brier_score          0.17    0.16    0.18    0.16    0.17    0.18    0.19
0.18    0.17    0.18
AUC                  0.90    0.92    0.88    0.88    0.89    0.87    0.86
0.89    0.89    0.90
Acc_by_package_fn    0.75    0.76    0.73    0.78    0.73    0.75    0.74
0.73    0.75    0.73

Metrics for Algorithm 2 (RF):
                     iter1   iter2   iter3   iter4   iter5   iter6   iter7
iter8   iter9  iter10
TP                   52.00   50.00   50.00   51.00   51.00   48.00   49.00
46.00   46.00   57.00
TN                  489.00  477.00  480.00  482.00  488.00  481.00  475.00
480.00  480.00  483.00
FP                    7.00   19.00   16.00   14.00    8.00   15.00   21.00
16.00   16.00   12.00
FN                   12.00   14.00   14.00   13.00   13.00   16.00   15.00
17.00   17.00    7.00
TPR                  0.81    0.78    0.78    0.80    0.80    0.75    0.77
0.73    0.73    0.89
TNR                  0.99    0.96    0.97    0.97    0.98    0.97    0.96
0.97    0.97    0.98
FPR                  0.01    0.04    0.03    0.03    0.02    0.03    0.04
0.03    0.03    0.02
FNR                  0.19    0.22    0.22    0.20    0.20    0.25    0.23
0.27    0.27    0.11
Precision            0.88    0.72    0.76    0.78    0.86    0.76    0.70
0.74    0.74    0.83
F1_measure           0.85    0.75    0.77    0.79    0.83    0.76    0.73
0.74    0.74    0.86
Accuracy             0.97    0.94    0.95    0.95    0.96    0.94    0.94
0.94    0.94    0.97
Error_rate           0.03    0.06    0.05    0.05    0.04    0.06    0.06
0.06    0.06    0.03
BACC                 0.90    0.87    0.87    0.88    0.89    0.86    0.86
```

```
0.85    0.85    0.93
TSS                 0.80    0.74    0.75    0.77    0.78    0.72    0.72
0.70    0.70    0.87
HSS                 0.83    0.72    0.74    0.76    0.81    0.72    0.69
0.70    0.70    0.84
Brier_score         0.04    0.05    0.04    0.04    0.03    0.05    0.05
0.04    0.05    0.04
AUC                 0.98    0.96    0.97    0.97    0.98    0.96    0.97
0.98    0.96    0.98
Acc_by_package_fn   0.97    0.94    0.95    0.95    0.96    0.94    0.94
0.94    0.94    0.97

Metrics for Algorithm 3 (LSTM):
                    iter1   iter2   iter3   iter4   iter5   iter6   iter7
iter8   iter9  iter10
TP                  50.00   52.00   50.00   51.00   52.00   45.00   49.00
46.00   46.00   48.00
TN                 483.00  463.00  461.00  455.00  477.00  474.00  467.00
478.00  467.00  460.00
FP                  13.00   33.00   35.00   41.00   19.00   22.00   29.00
18.00   29.00   35.00
FN                  14.00   12.00   14.00   13.00   12.00   19.00   15.00
17.00   17.00   16.00
TPR                 0.78    0.81    0.78    0.80    0.81    0.70    0.77
0.73    0.73    0.75
TNR                 0.97    0.93    0.93    0.92    0.96    0.96    0.94
0.96    0.94    0.93
FPR                 0.03    0.07    0.07    0.08    0.04    0.04    0.06
0.04    0.06    0.07
FNR                 0.22    0.19    0.22    0.20    0.19    0.30    0.23
0.27    0.27    0.25
Precision           0.79    0.61    0.59    0.55    0.73    0.67    0.63
0.72    0.61    0.58
F1_measure          0.79    0.70    0.67    0.65    0.77    0.69    0.69
0.72    0.67    0.65
Accuracy            0.95    0.92    0.91    0.90    0.94    0.93    0.92
0.94    0.92    0.91
Error_rate          0.05    0.08    0.09    0.10    0.06    0.07    0.08
0.06    0.08    0.09
BACC                0.88    0.87    0.86    0.86    0.89    0.83    0.85
0.85    0.84    0.84
TSS                 0.76    0.75    0.71    0.71    0.77    0.66    0.71
0.69    0.67    0.68
HSS                 0.76    0.65    0.62    0.60    0.74    0.65    0.65
0.69    0.62    0.60
Brier_score         0.04    0.06    0.07    0.08    0.05    0.06    0.06
0.05    0.06    0.07
AUC                 0.95    0.96    0.94    0.93    0.96    0.94    0.95
```

```
0.96    0.94    0.93
Acc_by_package_fn    0.95    0.92    0.91    0.90    0.94    0.93    0.92
0.94    0.92    0.91

Average performance over 10 folds (rounded):
                        KNN       RF     LSTM
TP                   58.700   50.000   48.900
TN                  358.800  481.500  468.500
FP                  137.100   14.400   27.400
FN                    5.100   13.800   14.900
TPR                   0.920    0.784    0.766
TNR                   0.724    0.971    0.945
FPR                   0.276    0.029    0.055
FNR                   0.080    0.216    0.234
Precision             0.300    0.778    0.649
F1_measure            0.453    0.780    0.700
Accuracy              0.746    0.950    0.924
Error_rate            0.254    0.050    0.076
BACC                  0.822    0.877    0.856
TSS                   0.644    0.754    0.711
HSS                   0.339    0.752    0.658
Brier_score           0.173    0.042    0.060
AUC                   0.890    0.972    0.945
Acc_by_package_fn     0.746    0.950    0.924
```

This code calculates and displays the performance metrics for the KNN, RF, and LSTM models over 10-fold cross-validation. It organizes the metrics for each fold into DataFrames, with each row representing an iteration and columns showing different evaluation metrics.

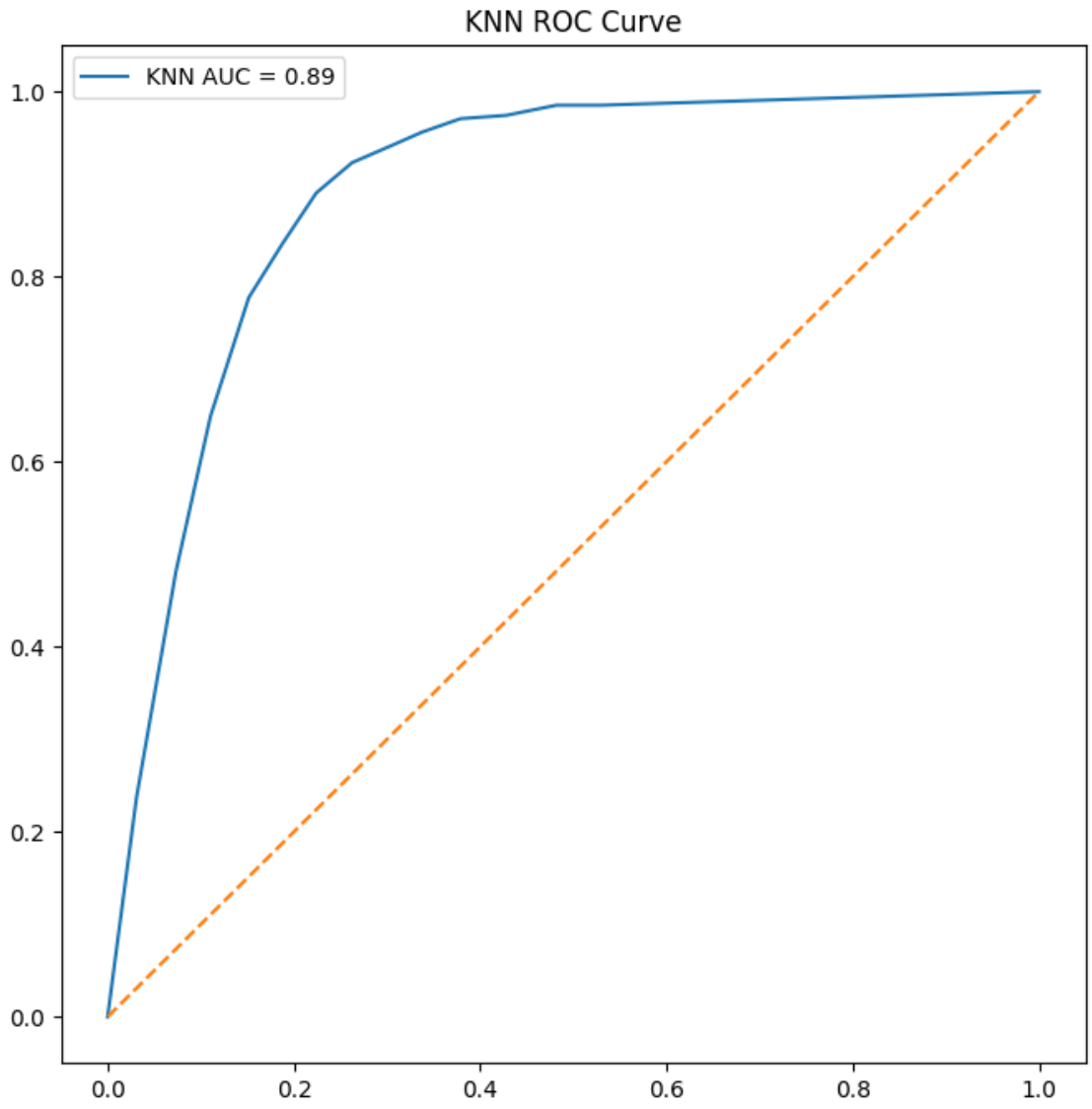
The metrics for each model are printed separately, showing how each performed in each fold. Then, it computes the average performance across all folds by averaging each column in the metric DataFrame. These averages are stored in a new DataFrame and printed, offering a summarized view of each model's overall performance. The results are rounded to three decimal places for clarity.

[14]:
```python
# 12. Final ROC Curves on Held-out Test Set

print("\nTraining final KNN on balanced data and plotting ROC...")
knn_model = KNeighborsClassifier(n_neighbors=best_n_neighbors, n_jobs=-1)
knn_model.fit(features_train_bal_all, labels_train_bal_all)
y_score = knn_model.predict_proba(features_test_all_std)[:, 1]
fpr, tpr, _ = roc_curve(labels_test_all, y_score)
roc_auc = auc(fpr, tpr)
plt.figure(figsize=(8, 8))
plt.plot(fpr, tpr, label=f"KNN AUC = {roc_auc:.2f}")
plt.plot([0, 1], [0, 1], linestyle="--")
plt.title("KNN ROC Curve")
plt.legend()
```

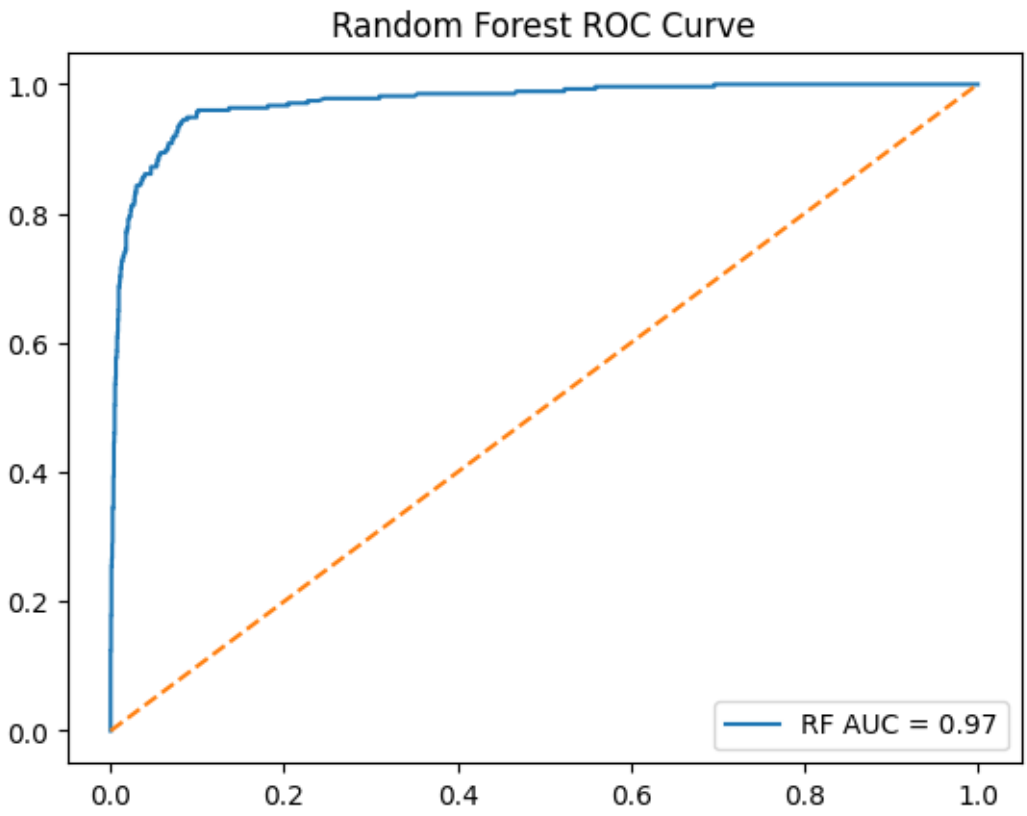
```
plt.savefig("roc_knn.png", bbox_inches="tight")
plt.show()

print(f"KNN Test AUC: {roc_auc:.4f}")

print("\nTraining final Random Forest on balanced data and plotting ROC...")
rf_model = RandomForestClassifier(
    min_samples_split=min_samples_split,
    n_estimators=n_estimators,
    n_jobs=-1
)
rf_model.fit(features_train_bal_all, labels_train_bal_all)
y_score_rf = rf_model.predict_proba(features_test_all_std)[:, 1]
fpr_rf, tpr_rf, _ = roc_curve(labels_test_all, y_score_rf)
roc_auc_rf = auc(fpr_rf, tpr_rf)
plt.figure()
plt.plot(fpr_rf, tpr_rf, label=f"RF AUC = {roc_auc_rf:.2f}")
plt.plot([0, 1], [0, 1], linestyle="--")
plt.title("Random Forest ROC Curve")
plt.legend()
plt.savefig("roc_rf.png", bbox_inches="tight")
plt.show()

print(f"Random Forest Test AUC: {roc_auc_rf:.4f}")

print("\nTraining final LSTM on balanced data and plotting ROC...")
lstm_model = Sequential()
lstm_model.add(
    LSTM(
        64,
        activation='relu',
        input_shape=(n_features, 1),
        return_sequences=False
    )
)
lstm_model.add(Dense(1, activation='sigmoid'))
lstm_model.compile(loss='binary_crossentropy', optimizer='adam',␣
 ↪metrics=['accuracy'])

X_train_array = np.array(features_train_bal_all)
X_test_array = features_test_all_std.to_numpy()
y_train_array = np.array(labels_train_bal_all)
y_test_array = labels_test_all.to_numpy()

input_train = X_train_array.reshape(len(X_train_array), n_features, 1)
input_test = X_test_array.reshape(len(X_test_array), n_features, 1)
```

```
lstm_model.fit(input_train, y_train_array, epochs=50,
               validation_data=(input_test, y_test_array), verbose=0)

predict_lstm = lstm_model.predict(input_test, verbose=0)
fpr_lstm, tpr_lstm, _ = roc_curve(labels_test_all, predict_lstm)
roc_auc_lstm = auc(fpr_lstm, tpr_lstm)

plt.figure()
plt.plot(fpr_lstm, tpr_lstm, label=f"LSTM AUC = {roc_auc_lstm:.2f}")
plt.plot([0, 1], [0, 1], linestyle="--")
plt.title("LSTM ROC Curve")
plt.legend()
plt.savefig("roc_lstm.png", bbox_inches="tight")
plt.show()

print(f"LSTM Test AUC: {roc_auc_lstm:.4f}")
```

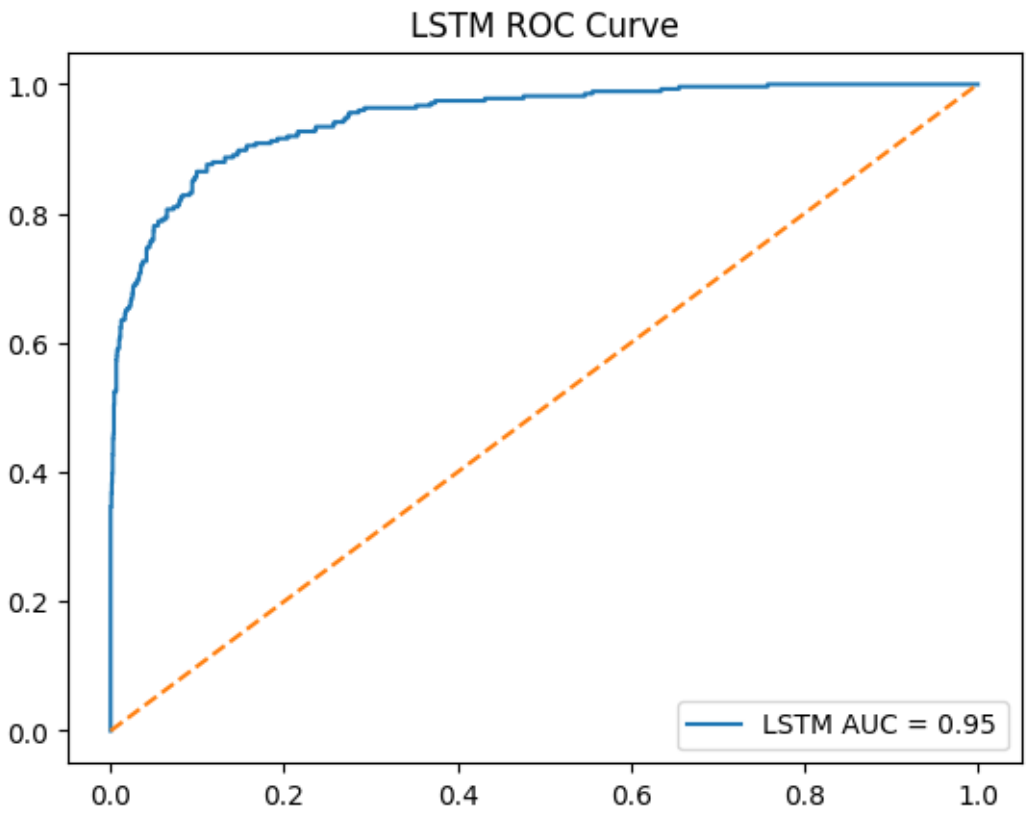
```
Training final KNN on balanced data and plotting ROC…
```




```
KNN Test AUC: 0.8895

Training final Random Forest on balanced data and plotting ROC…
```




```
Random Forest Test AUC: 0.9730

Training final LSTM on balanced data and plotting ROC…
```




```
LSTM Test AUC: 0.9487
```

This code trains and evaluates the KNN, Random Forest, and LSTM models on the balanced training data, plotting their ROC curves and computing the AUC for each.

For KNN, the model is trained on the balanced data, and predicted probabilities are used to calculate the FPR and TPR for the ROC curve. The AUC is computed, and the ROC curve is saved as "roc_knn.png". The AUC value is printed.

Similarly, the Random Forest model is trained on the balanced data, with its ROC curve plotted and saved as "roc_rf.png", and its AUC printed.

Finally, the LSTM model is trained on the reshaped balanced data. Its ROC curve is plotted and saved as "roc_lstm.png", and the AUC is printed. Each model's performance is summarized by their AUC scores, reflecting their prediction quality on the test set.