

# nakarmi\_avina\_finaltermproj

November 20, 2024

## 1 GitHub Repository

[https://github.com/avinanakarmi/nakarmi\\_avina\\_finaltermproj](https://github.com/avinanakarmi/nakarmi_avina_finaltermproj)

## 2 System Requirement

- Python version: 3.12

## 3 Procedure to Run the Project

To run the project and generate association rules, follow these steps:

### 1. Clone or Download the Project Files:

First, clone the repository or download the project files to your local machine.

Ensure that all necessary scripts and datasets are present in the project folder.

```
git clone https://github.com/avinanakarmi/nakarmi_avina_finaltermproj.git
cd nakarmi_avina_finaltermproj
```

### 2. Install the Required Libraries:

Make sure your python version is 3.12 for compatibility with tensorflow. If you haven't installed the libraries listed in the prerequisites section, you can do so by running:

```
pip install -r nakarmi_avina_requirements.txt
```

### 3. Run the Project:

Open a terminal or command prompt in the project directory and run:

```
python nakarmi_avina_finaltermproj.py
```

### 6. View Results:

Initially the program could take a while to load and transform the datasets without any prompt on the console. On running the .py script, visualization are displayed in a pop-up window that need to be closed after observing to enable the program to continue working. Once the script finishes running, it will display the performance metrics for selected models.

### 7. Evaluate Performance:

The report analyses all relevant performance metrics given the property of dataset and recommends the best model.

## 4 Objective

The objective of this project was to develop and evaluate three machine learning models—Random Forest, Decision Tree, and a 1D Convolutional Neural Network (CNN)—to classify data from a large, imbalanced dataset with high multicollinearity. The goal was to calculate and analyze a

comprehensive set of performance metrics, including true positives (TP), true negatives (TN), false positives (FP), false negatives (FN), true positive rate (TPR), true negative rate (TNR), precision, negative predictive value (NPV), false positive rate (FPR), false discovery rate (FDR), false negative rate (FNR), accuracy (ACC), F1 score, error rate, balanced accuracy (BACC), true skill statistic (TSS), Heidke skill score (HSS), Brier score (BS), and area under the ROC curve (AUC). By comparing these metrics, the aim was to identify and recommend the best-performing model for accurately classifying the dataset. This evaluation considered both predictive performance and robustness to data imbalances and feature correlations.

```
[2]: import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
```

## 5 Data Exploration and Preprocessing

### 5.1 Data Description

The dataset used in this project is a combination of two datasets, related to red and white vinho verde wine samples, from the north of Portugal. The goal is to recommend wine based on physicochemical tests.

```
[3]: data = pd.read_csv("./nakarmi_avina_finaltermproj.csv", sep=";")

data.describe()
```

```
[3]:
```

	fixed acidity	volatile acidity	citric acid	residual sugar	\
count	6497.000000	6497.000000	6497.000000	6497.000000	
mean	7.215307	0.339666	0.318633	5.443235	
std	1.296434	0.164636	0.145318	4.757804	
min	3.800000	0.080000	0.000000	0.600000	
25%	6.400000	0.230000	0.250000	1.800000	
50%	7.000000	0.290000	0.310000	3.000000	
75%	7.700000	0.400000	0.390000	8.100000	
max	15.900000	1.580000	1.660000	65.800000	

	chlorides	free sulfur dioxide	total sulfur dioxide	density	\
count	6497.000000	6497.000000	6497.000000	6497.000000	
mean	0.056034	30.525319	115.744574	0.994697	
std	0.035034	17.749400	56.521855	0.002999	
min	0.009000	1.000000	6.000000	0.987110	
25%	0.038000	17.000000	77.000000	0.992340	
50%	0.047000	29.000000	118.000000	0.994890	
75%	0.065000	41.000000	156.000000	0.996990	
max	0.611000	289.000000	440.000000	1.038980	

	pH	sulphates	alcohol	quality
count	6497.000000	6497.000000	6497.000000	6497.000000

mean	3.218501	0.531268	10.491801	5.818378
std	0.160787	0.148806	1.192712	0.873255
min	2.720000	0.220000	8.000000	3.000000
25%	3.110000	0.430000	9.500000	5.000000
50%	3.210000	0.510000	10.300000	6.000000
75%	3.320000	0.600000	11.300000	6.000000
max	4.010000	2.000000	14.900000	9.000000

```
[4]: data.tail()
```

```
[4]:      fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  \
6492           6.2           0.21           0.29           1.6           0.039
6493           6.6           0.32           0.36           8.0           0.047
6494           6.5           0.24           0.19           1.2           0.041
6495           5.5           0.29           0.30           1.1           0.022
6496           6.0           0.21           0.38           0.8           0.020
```

	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	\
6492	24.0	92.0	0.99114	3.27	0.50	
6493	57.0	168.0	0.99490	3.15	0.46	
6494	30.0	111.0	0.99254	2.99	0.46	
6495	20.0	110.0	0.98869	3.34	0.38	
6496	22.0	98.0	0.98941	3.26	0.32	

	alcohol	quality
6492	11.2	6
6493	9.6	5
6494	9.4	6
6495	12.8	7
6496	11.8	6

## 5.2 Data transformation

The project requires us to work with binary classification data, whereas this data set has multiclass classification for the target attribute quality. To align with the project requirements a “recommended” attribute is derived from the “quality” attribute. If the quality of data item is higher than 6, the wine is recommended i.e, the recommended attribute has value 1.

```
[5]: data["recommendation"] = (data["quality"] > 6).astype('int32')
data = data.drop("quality", axis=1)
```

```
[6]: data.tail()
```

```
[6]:      fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  \
6492           6.2           0.21           0.29           1.6           0.039
6493           6.6           0.32           0.36           8.0           0.047
6494           6.5           0.24           0.19           1.2           0.041
6495           5.5           0.29           0.30           1.1           0.022
```

6496	6.0	0.21	0.38	0.8	0.020
------	-----	------	------	-----	-------

	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates \
6492	24.0	92.0	0.99114	3.27	0.50
6493	57.0	168.0	0.99490	3.15	0.46
6494	30.0	111.0	0.99254	2.99	0.46
6495	20.0	110.0	0.98869	3.34	0.38
6496	22.0	98.0	0.98941	3.26	0.32

	alcohol	recommendation
6492	11.2	0
6493	9.6	0
6494	9.4	0
6495	12.8	1
6496	11.8	0

### 5.3 Type of attributes and null values

```
[7]: print("Dataframe shape", data.shape)
print()
print("Check type of data")
print(data.dtypes)
print()
print("Check for na")
print(data.isna().sum())
```

Dataframe shape (6497, 12)

Check type of data

fixed acidity	float64
volatile acidity	float64
citric acid	float64
residual sugar	float64
chlorides	float64
free sulfur dioxide	float64
total sulfur dioxide	float64
density	float64
pH	float64
sulphates	float64
alcohol	float64
recommendation	int32
dtype:	object

Check for na

fixed acidity	0
volatile acidity	0
citric acid	0
residual sugar	0

```

chlorides          0
free sulfur dioxide 0
total sulfur dioxide 0
density            0
pH                0
sulphates          0
alcohol            0
recommendation     0
dtype: int64

```

## 5.4 Target Class Distribution

The dataset contains 19.7% positive classes (recommended = 1) and 80.3% negative classes (recommended = 0). While the imbalance is not extreme, the difference in target class distribution is significant, given that the dataset contains only 6,497 records.

To address class imbalance, several strategies can be considered:

1. **Data Augmentation:** Generate additional samples for the underrepresented class.
2. **Stratified Splitting:** Ensure that both training and testing datasets are representative of the target class distribution.
3. **Class Weight Adjustment:** Assign higher weights to the underrepresented class during model training.

Since the primary objective of the project was to evaluate model performance rather than directly addressing class imbalance, the dataset was not explicitly balanced. Instead, stratified k-fold cross-validation was employed to ensure that each fold in the cross-validation process contained a representative subset of the target class distribution. Additionally, for experimentation, class weights were adjusted for some selected models to account for the imbalance.

```

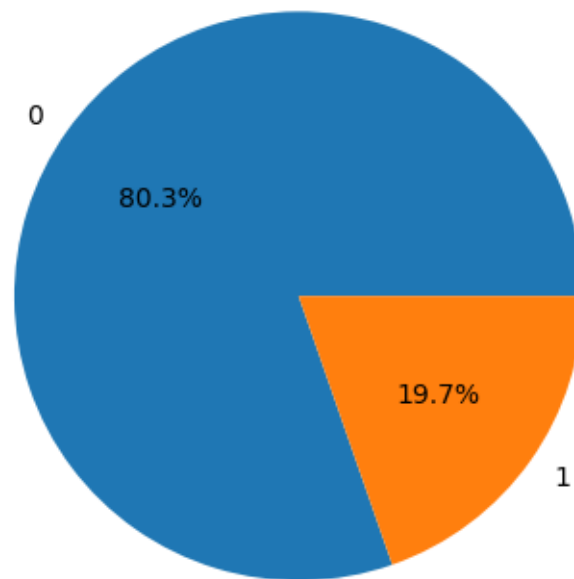
[8]: class_dist = data["recommendation"].value_counts().sort_index()
plt.pie(class_dist.values, labels=class_dist.index, autopct='%1.1f%%')

plt.title("Check for imbalanced data")

plt.show()

```

## Check for imbalanced data



### 5.5 Attribute collinearity Analysis

As suggested by the correlation matrix heatmap, there are moderate to strong positive and negative correlations between several pairs of features. (Fixed Acidity, Citric Acid), (Free Sulfur Dioxide, Total Sulfur Dioxide), and (Density, Residual Sugar) have high positive correlation. (Alcohol, Density), (Alcohol, Residual Sugar) and (Residual Sugar, pH) have high negative correlation. The high correlations indicate potential multicollinearity, which could affect model performance and interpretation.

The Variance Inflation Factor (VIF) table quantifies the multicollinearity among features, with a VIF value above 10 typically indicating high multicollinearity. High VIF values of density, pH, alcohol, and fixed acidity suggest multicollinearity issues.

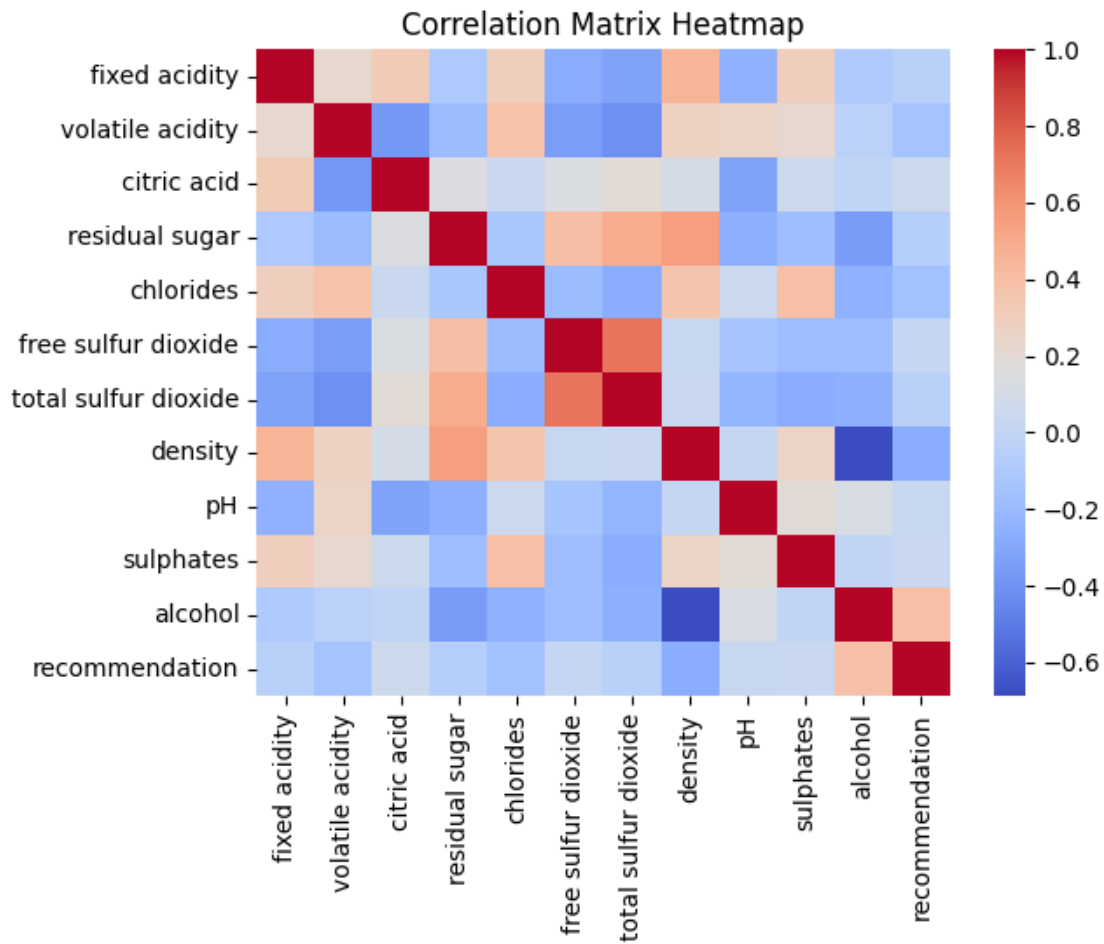
Both the heatmap and VIF values indicate high multicollinearity among several features, especially Density, pH, and Alcohol. This may impact model interpretability and could lead to issues in certain machine learning models that are sensitive to multicollinearity.

```
[9]: corr_matrix = data.corr(numeric_only = True)

sns.heatmap(corr_matrix, annot=False, cmap='coolwarm')

plt.title('Correlation Matrix Heatmap')
```

```
plt.show()
```



```
[10]: from statsmodels.stats.outliers_influence import variance_inflation_factor
```

```
X = data.drop(columns=['recommendation'])
```

```
vif_data = pd.DataFrame()
```

```
vif_data['Feature'] = X.columns
```

```
vif_data['VIF'] = [variance_inflation_factor(X.values, i) for i in range(X.  
↪shape[1])]
```

```
print(vif_data)
```

	Feature	VIF
0	fixed acidity	58.897405
1	volatile acidity	8.943681
2	citric acid	9.340251
3	residual sugar	3.576148

4	chlorides	5.575434
5	free sulfur dioxide	8.452180
6	total sulfur dioxide	14.732237
7	density	936.984064
8	pH	589.005172
9	sulphates	18.491253
10	alcohol	107.135452

## 6 Model Selection

1. **RandomForest:** Given the high Variance Inflation Factor (VIF) values for features like “density” and “pH,” Random Forest can mitigate the risk of overfitting caused by correlated features by averaging across multiple decision trees. As indicated by the pie chart (80.3% for one class and 19.7% for the other), the dataset chosen for this project is imbalanced. Random Forest tends to be more resilient with imbalanced data when class weights are adjusted.
2. **Decision Trees:** The correlation matrix shows that the features have a complex relationship. Decision Trees can capture non-linear relationships between features and the target class.
3. **Conv1D:** Given the high VIF values, as the CNN can learn more robust representations of the data, minimizing multicollinearity’s effects. CNNs can also be fine-tuned with techniques like class weights, which helps the model focus on minority classes.

### 6.1 Random Forest

```
[11]: from sklearn.ensemble import RandomForestClassifier

# adjusting class weights in this model
rf = RandomForestClassifier(class_weight="balanced")

def predict_with_random_forest(X_train, y_train, X_test):
    rf.fit(X_train, y_train)

    y_pred = rf.predict(X_test)
    y_prob = rf.predict_proba(X_test)
    return y_pred, y_prob[:,1]
```

### 6.2 Decision tree

```
[12]: from sklearn.tree import DecisionTreeClassifier

dt = DecisionTreeClassifier()
def predict_with_decision_tree(X_train, y_train, X_test):
    dt.fit(X_train, y_train)

    y_pred = dt.predict(X_test)
```



```

y_prob = dt.predict_proba(X_test)
return y_pred, y_prob[:,1]

```

### 6.3 Conv1D

```

[13]: from tensorflow.keras.models import Sequential
      from tensorflow.keras.layers import Conv1D, MaxPooling1D, Flatten, Dense,
      ↳Dropout, Input
      from tensorflow.keras.optimizers import Adam
      from sklearn.utils.class_weight import compute_class_weight

model = None
def predict_with_conv1d(X_train, y_train, X_test):
    global model

    # Calculate class weights based on the training labels
    class_weights = compute_class_weight(class_weight='balanced', classes=np.
    ↳unique(y_train), y=y_train)
    class_weights = {i: weight for i, weight in enumerate(class_weights)}

    if model is None:
        model = Sequential()
        model.add(Input(shape= (X_train.shape[1], 1)))
        model.add(Conv1D(filters=32, kernel_size=2, activation='relu'))
        model.add(Dropout(0.2))
        model.add(Flatten())
        model.add(Dense(64, activation='relu'))
        model.add(Dense(1, activation='sigmoid'))
        model.compile(optimizer=Adam(learning_rate=0.001),
        ↳loss='binary_crossentropy', metrics=['accuracy'])
        # Fit the model with class weights
        history = model.fit(X_train.values, y_train.values, epochs=10, batch_size=32,
        ↳verbose=0, class_weight=class_weights)
        y_pred = model.predict(X_test)
        return y_pred

```

## 7 Util Functions

```

[14]: from typing import TypedDict

class Measures(TypedDict):
    tp: int
    tn: int
    fp: int
    fn: int
    tpr: float

```

```

tnr: float
precision: float
npv: float
fpr: float
fdr: float
fnr: float
acc: float
f1: float
err_rate: float
bacc: float
tss: float
hss: float
bss: float
auc: float

```

```

[15]: from sklearn.metrics import confusion_matrix

def get_classification_outcomes(y_true, y_pred):
    tn, fp, fn, tp = confusion_matrix(y_true, y_pred).ravel()
    return tn, fp, fn, tp

```

```

[16]: def safe_divide(numerator, denominator):
        if denominator == 0:
            return 0
        else:
            return numerator / denominator

```

```

[17]: def find_auc(y_true, y_prob):
    df = pd.DataFrame({"actual": y_true, "probability": y_prob})
    df = df.sort_values(by="probability", ascending=True)
    TPR = []
    FPR = []
    for i in range(len(y_prob)):
        df["predicted"] = np.hstack([np.zeros(i), np.ones(len(y_prob) - i)])
        tp = sum((df["actual"] == 1) & (df["predicted"] == 1))
        fp = sum((df["actual"] == 0) & (df["predicted"] == 1))
        tn = sum((df["actual"] == 0) & (df["predicted"] == 0))
        fn = sum((df["actual"] == 1) & (df["predicted"] == 0))
        tpr = safe_divide(tp, (tp + fn))
        fpr = safe_divide(fp, (tn + fp))
        TPR.append(tpr)
        FPR.append(fpr)

    auc = np.abs(np.trapezoid(TPR, FPR))
    return auc

```

```
[18]: def calc_bss(y_true, bs):
    # BS_ref requires a reference model to compare the performance
    return 1 - safe_divide(bs, bs_ref)

[19]: from sklearn.metrics import brier_score_loss

def calculate_measures(y_true, y_pred, y_prob) -> Measures:
    measures = {}
    tn, fp, fn, tp = get_classification_outcomes(y_true, y_pred)
    p = tp + fn
    n = tn + fp
    measures['tp'] = tp
    measures['tn'] = tn
    measures['fp'] = fp
    measures['fn'] = fn
    measures['tpr'] = safe_divide(tp, p)
    measures['tnr'] = safe_divide(tn, n)
    measures['precision'] = safe_divide(tp, (fp + tp))
    measures['npv'] = safe_divide(tn, (tn + fn))
    measures['fpr'] = safe_divide(fp, n)
    measures['fdr'] = safe_divide(fp, (fp + tp))
    measures['fnr'] = safe_divide(fn, p)
    measures['acc'] = safe_divide((tp + tn), (p + n))
    measures['f1'] = safe_divide((2 * measures['precision'] * measures['tpr']),
    ↪(measures['precision'] + measures['tpr']))
    measures['err_rate'] = safe_divide((fp + fn), (p + n))
    measures['bacc'] = (measures['tpr'] + measures['tnr']) / 2
    measures['tss'] = (safe_divide(tp, (fn + tp))) - (safe_divide(fp, (fp + tn)))
    measures['hss'] = safe_divide(2 * ((tp * tn) - (fp * fn)), ((tp + fn) * (fn +
    ↪tn) + (tp + fp) * (fp + tn)))

    measures['bs'] = brier_score_loss(y_true = y_true, y_proba = y_prob)
    # measures['bss'] = calc_bss(y_true, measures['bs'])
    measures['auc'] = find_auc(y_true, y_prob)

    return Measures(measures)

[33]: ##### Visualize measure in each fold
from typing import Dict, List

def viz_measures_k_fold(k, **kwargs: Measures):
    suffix = 'th'
    if k%10 == 1: suffix = 'st'
    elif k%10 == 2: suffix = 'nd'
    elif k%10 == 3: suffix = 'rd'
    print()
```

```

print('Visualizing Model Performance', f'in {k}{suffix} fold:' if k > 0 else '
↳')
print(f"{'Measure':<13}", end='')
for model in kwargs.keys():
    print(f'{model:<13}', end='')
print()
tup = next(iter(kwargs.items()))
for measure in tup[1].keys():
    print(f'{measure:<13}', end='')
    for _, measures in kwargs.items():
        print(f'{measures[measure]:<13.2f}', end='')
    print()
print()

def viz_measures_model(model, measures: List[Measures]):
    print()
    print('Visualizing', model, 'Performance in Each Fold')

    # Create a DataFrame from the measures
    measure_names = measures[0].keys()
    data = {measure: [k_measures[measure] for k_measures in measures] for measure
↳in measure_names}
    df = pd.DataFrame(data)
    df.index = [f"{i+1}" for i in range(len(measures))]

    # Print the DataFrame
    print(df)
    print()

```

## 8 Train and test dataset preparation

```

[21]: from sklearn.model_selection import train_test_split

y = data["recommendation"]
X = data.drop("recommendation", axis=1)
data_X_train, data_X_test, data_y_train, data_y_test = train_test_split(X, y,
↳test_size=0.33, random_state=42)

```

## 9 Model Metrics Calculation

```

[22]: from sklearn.model_selection import StratifiedKFold

rf_measures = []
dt_measures = []
conv1D_measures = []

```

```

### Ensures each fold has the same proportion of classes as the complete
dataset.
kf = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)

for idx, (train_index, test_index) in enumerate(kf.split(data_X_train,
data_y_train), start = 1):
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    rf_pred, rf_prob = predict_with_random_forest(X_train, y_train, X_test)
    measures = calculate_measures(y_test, rf_pred, rf_prob)
    rf_measures.append(measures)

    dt_pred, dt_prob = predict_with_decision_tree(X_train, y_train, X_test)
    measures = calculate_measures(y_test, dt_pred, dt_prob)
    dt_measures.append(measures)

    conv1d_prob = predict_with_conv1d(X_train, y_train, X_test)
    conv1d_pred = (conv1d_prob > 0.5).astype(int)
    measures = calculate_measures(y_test, [item for row in conv1d_pred for item
in row], conv1d_prob.flatten())
    conv1D_measures.append(measures)

    viz_measures_k_fold(idx, RandomForest = rf_measures[idx - 1],
DecisionTree=dt_measures[idx - 1], Conv1D=conv1D_measures[idx - 1])

```

14/14

0s 1ms/step

Visualizing Model Performance in 1st fold:

Measure	RandomForest	DecisionTree	Conv1D
tp	39.00	45.00	58.00
tn	354.00	328.00	169.00
fp	9.00	35.00	194.00
fn	34.00	28.00	15.00
tpr	0.53	0.62	0.79
tnr	0.98	0.90	0.47
precision	0.81	0.56	0.23
npv	0.91	0.92	0.92
fpr	0.02	0.10	0.53
fdr	0.19	0.44	0.77
fnr	0.47	0.38	0.21
acc	0.90	0.86	0.52
f1	0.64	0.59	0.36
err_rate	0.10	0.14	0.48
bacc	0.75	0.76	0.63

tss	0.51	0.52	0.26
hss	0.59	0.50	0.13
bs	0.08	0.14	0.25
auc	0.91	0.73	0.71

14/14                      0s 254us/step

Visualizing Model Performance in 2nd fold:

Measure	RandomForest	DecisionTree	Conv1D
tp	47.00	48.00	55.00
tn	354.00	330.00	246.00
fp	5.00	29.00	113.00
fn	30.00	29.00	22.00
tpr	0.61	0.62	0.71
tnr	0.99	0.92	0.69
precision	0.90	0.62	0.33
npv	0.92	0.92	0.92
fpr	0.01	0.08	0.31
fdr	0.10	0.38	0.67
fnr	0.39	0.38	0.29
acc	0.92	0.87	0.69
f1	0.73	0.62	0.45
err_rate	0.08	0.13	0.31
bacc	0.80	0.77	0.70
tss	0.60	0.54	0.40
hss	0.68	0.54	0.27
bs	0.06	0.13	0.19
auc	0.94	0.81	0.78

14/14                      0s 273us/step

Visualizing Model Performance in 3rd fold:

Measure	RandomForest	DecisionTree	Conv1D
tp	41.00	46.00	66.00
tn	352.00	323.00	124.00
fp	9.00	38.00	237.00
fn	33.00	28.00	8.00
tpr	0.55	0.62	0.89
tnr	0.98	0.89	0.34
precision	0.82	0.55	0.22
npv	0.91	0.92	0.94
fpr	0.02	0.11	0.66
fdr	0.18	0.45	0.78
fnr	0.45	0.38	0.11
acc	0.90	0.85	0.44
f1	0.66	0.58	0.35
err_rate	0.10	0.15	0.56
bacc	0.76	0.76	0.62

tss	0.53	0.52	0.24
hss	0.61	0.49	0.11
bs	0.08	0.15	0.28
auc	0.91	0.78	0.77

14/14                      0s 275us/step

Visualizing Model Performance in 4th fold:

Measure	RandomForest	DecisionTree	Conv1D
tp	45.00	55.00	66.00
tn	336.00	315.00	263.00
fp	13.00	34.00	86.00
fn	41.00	31.00	20.00
tpr	0.52	0.64	0.77
tnr	0.96	0.90	0.75
precision	0.78	0.62	0.43
npv	0.89	0.91	0.93
fpr	0.04	0.10	0.25
fdr	0.22	0.38	0.57
fnr	0.48	0.36	0.23
acc	0.88	0.85	0.76
f1	0.62	0.63	0.55
err_rate	0.12	0.15	0.24
bacc	0.74	0.77	0.76
tss	0.49	0.54	0.52
hss	0.55	0.54	0.40
bs	0.08	0.15	0.17
auc	0.92	0.75	0.82

14/14                      0s 291us/step

Visualizing Model Performance in 5th fold:

Measure	RandomForest	DecisionTree	Conv1D
tp	35.00	38.00	46.00
tn	364.00	341.00	302.00
fp	7.00	30.00	69.00
fn	29.00	26.00	18.00
tpr	0.55	0.59	0.72
tnr	0.98	0.92	0.81
precision	0.83	0.56	0.40
npv	0.93	0.93	0.94
fpr	0.02	0.08	0.19
fdr	0.17	0.44	0.60
fnr	0.45	0.41	0.28
acc	0.92	0.87	0.80
f1	0.66	0.58	0.51
err_rate	0.08	0.13	0.20
bacc	0.76	0.76	0.77

tss	0.53	0.51	0.53
hss	0.62	0.50	0.40
bs	0.07	0.13	0.16
auc	0.93	0.68	0.83

14/14                      0s 280us/step

Visualizing Model Performance in 6th fold:

Measure	RandomForest	DecisionTree	Conv1D
tp	48.00	53.00	62.00
tn	350.00	315.00	258.00
fp	9.00	44.00	101.00
fn	28.00	23.00	14.00
tpr	0.63	0.70	0.82
tnr	0.97	0.88	0.72
precision	0.84	0.55	0.38
npv	0.93	0.93	0.95
fpr	0.03	0.12	0.28
fdr	0.16	0.45	0.62
fnr	0.37	0.30	0.18
acc	0.91	0.85	0.74
f1	0.72	0.61	0.52
err_rate	0.09	0.15	0.26
bacc	0.80	0.79	0.77
tss	0.61	0.57	0.53
hss	0.67	0.52	0.37
bs	0.07	0.15	0.19
auc	0.93	0.76	0.85

14/14                      0s 281us/step

Visualizing Model Performance in 7th fold:

Measure	RandomForest	DecisionTree	Conv1D
tp	47.00	51.00	71.00
tn	344.00	322.00	258.00
fp	7.00	29.00	93.00
fn	37.00	33.00	13.00
tpr	0.56	0.61	0.85
tnr	0.98	0.92	0.74
precision	0.87	0.64	0.43
npv	0.90	0.91	0.95
fpr	0.02	0.08	0.26
fdr	0.13	0.36	0.57
fnr	0.44	0.39	0.15
acc	0.90	0.86	0.76
f1	0.68	0.62	0.57
err_rate	0.10	0.14	0.24
bacc	0.77	0.76	0.79



tss	0.54	0.52	0.58
hss	0.62	0.53	0.43
bs	0.08	0.14	0.18
auc	0.94	0.74	0.85

14/14                      0s 264us/step

Visualizing Model Performance in 8th fold:

Measure	RandomForest	DecisionTree	Conv1D
tp	49.00	55.00	82.00
tn	340.00	312.00	199.00
fp	4.00	32.00	145.00
fn	42.00	36.00	9.00
tpr	0.54	0.60	0.90
tnr	0.99	0.91	0.58
precision	0.92	0.63	0.36
npv	0.89	0.90	0.96
fpr	0.01	0.09	0.42
fdr	0.08	0.37	0.64
fnr	0.46	0.40	0.10
acc	0.89	0.84	0.65
f1	0.68	0.62	0.52
err_rate	0.11	0.16	0.35
bacc	0.76	0.76	0.74
tss	0.53	0.51	0.48
hss	0.62	0.52	0.31
bs	0.08	0.16	0.23
auc	0.93	0.72	0.83

14/14                      0s 260us/step

Visualizing Model Performance in 9th fold:

Measure	RandomForest	DecisionTree	Conv1D
tp	45.00	50.00	77.00
tn	341.00	323.00	185.00
fp	8.00	26.00	164.00
fn	41.00	36.00	9.00
tpr	0.52	0.58	0.90
tnr	0.98	0.93	0.53
precision	0.85	0.66	0.32
npv	0.89	0.90	0.95
fpr	0.02	0.07	0.47
fdr	0.15	0.34	0.68
fnr	0.48	0.42	0.10
acc	0.89	0.86	0.60
f1	0.65	0.62	0.47
err_rate	0.11	0.14	0.40
bacc	0.75	0.75	0.71

tss	0.50	0.51	0.43
hss	0.58	0.53	0.25
bs	0.09	0.14	0.25
auc	0.91	0.70	0.80

14/14                      0s 267us/step

Visualizing Model Performance in 10th fold:

Measure	RandomForest	DecisionTree	Conv1D
tp	35.00	47.00	48.00
tn	358.00	327.00	286.00
fp	7.00	38.00	79.00
fn	35.00	23.00	22.00
tpr	0.50	0.67	0.69
tnr	0.98	0.90	0.78
precision	0.83	0.55	0.38
npv	0.91	0.93	0.93
fpr	0.02	0.10	0.22
fdr	0.17	0.45	0.62
fnr	0.50	0.33	0.31
acc	0.90	0.86	0.77
f1	0.62	0.61	0.49
err_rate	0.10	0.14	0.23
bacc	0.74	0.78	0.73
tss	0.48	0.57	0.47
hss	0.57	0.52	0.35
bs	0.07	0.14	0.14
auc	0.94	0.78	0.86

```
[34]: viz_measures_model("Random Forest", rf_measures)
```

Visualizing Random Forest Performance in Each Fold

	tp	tn	fp	fn	tpr	tnr	precision	npv	fpr	\
1	39	354	9	34	0.534247	0.975207	0.812500	0.912371	0.024793	
2	47	354	5	30	0.610390	0.986072	0.903846	0.921875	0.013928	
3	41	352	9	33	0.554054	0.975069	0.820000	0.914286	0.024931	
4	45	336	13	41	0.523256	0.962751	0.775862	0.891247	0.037249	
5	35	364	7	29	0.546875	0.981132	0.833333	0.926209	0.018868	
6	48	350	9	28	0.631579	0.974930	0.842105	0.925926	0.025070	
7	47	344	7	37	0.559524	0.980057	0.870370	0.902887	0.019943	
8	49	340	4	42	0.538462	0.988372	0.924528	0.890052	0.011628	
9	45	341	8	41	0.523256	0.977077	0.849057	0.892670	0.022923	
10	35	358	7	35	0.500000	0.980822	0.833333	0.910941	0.019178	

	fdr	fnr	acc	f1	err_rate	bacc	tss	\
1	0.187500	0.465753	0.901376	0.644628	0.098624	0.754727	0.509453	

2	0.096154	0.389610	0.919725	0.728682	0.080275	0.798231	0.596462
3	0.180000	0.445946	0.903448	0.661290	0.096552	0.764562	0.529123
4	0.224138	0.476744	0.875862	0.625000	0.124138	0.743003	0.486007
5	0.166667	0.453125	0.917241	0.660377	0.082759	0.764004	0.528007
6	0.157895	0.368421	0.914943	0.721805	0.085057	0.803255	0.606509
7	0.129630	0.440476	0.898851	0.681159	0.101149	0.769790	0.539581
8	0.075472	0.461538	0.894253	0.680556	0.105747	0.763417	0.526834
9	0.150943	0.476744	0.887356	0.647482	0.112644	0.750167	0.500333
10	0.166667	0.500000	0.903448	0.625000	0.096552	0.740411	0.480822

	hss	bs	auc
1	0.590190	0.080552	0.905242
2	0.683639	0.064562	0.942445
3	0.607434	0.076150	0.910272
4	0.553965	0.082919	0.924102
5	0.615555	0.065188	0.931941
6	0.672806	0.070486	0.931755
7	0.624397	0.076919	0.936610
8	0.622410	0.075102	0.932373
9	0.584899	0.085250	0.913240
10	0.573529	0.066119	0.940509

```
[35]: viz_measures_model("Decision Tree", dt_measures)
```

Visualizing Decision Tree Performance in Each Fold

	tp	tn	fp	fn	tpr	tnr	precision	npv	fpr \
1	45	328	35	28	0.616438	0.903581	0.562500	0.921348	0.096419
2	48	330	29	29	0.623377	0.919220	0.623377	0.919220	0.080780
3	46	323	38	28	0.621622	0.894737	0.547619	0.920228	0.105263
4	55	315	34	31	0.639535	0.902579	0.617978	0.910405	0.097421
5	38	341	30	26	0.593750	0.919137	0.558824	0.929155	0.080863
6	53	315	44	23	0.697368	0.877437	0.546392	0.931953	0.122563
7	51	322	29	33	0.607143	0.917379	0.637500	0.907042	0.082621
8	55	312	32	36	0.604396	0.906977	0.632184	0.896552	0.093023
9	50	323	26	36	0.581395	0.925501	0.657895	0.899721	0.074499
10	47	327	38	23	0.671429	0.895890	0.552941	0.934286	0.104110

	fdr	fnr	acc	f1	err_rate	bacc	tss \
1	0.437500	0.383562	0.855505	0.588235	0.144495	0.760010	0.520020
2	0.376623	0.376623	0.866972	0.623377	0.133028	0.771298	0.542597
3	0.452381	0.378378	0.848276	0.582278	0.151724	0.758179	0.516358
4	0.382022	0.360465	0.850575	0.628571	0.149425	0.771057	0.542114
5	0.441176	0.406250	0.871264	0.575758	0.128736	0.756444	0.512887
6	0.453608	0.302632	0.845977	0.612717	0.154023	0.787403	0.574806
7	0.362500	0.392857	0.857471	0.621951	0.142529	0.762261	0.524522
8	0.367816	0.395604	0.843678	0.617978	0.156322	0.755686	0.511372

9	0.342105	0.418605	0.857471	0.617284	0.142529	0.753448	0.506897
10	0.447059	0.328571	0.859770	0.606452	0.140230	0.783659	0.567319

	hss	bs	auc
1	0.500836	0.144495	0.731235
2	0.542597	0.133028	0.806172
3	0.490035	0.151724	0.775249
4	0.535081	0.149425	0.746152
5	0.499959	0.128736	0.680972
6	0.518352	0.154023	0.760226
7	0.534197	0.142529	0.739791
8	0.519774	0.156322	0.719716
9	0.530123	0.142529	0.704738
10	0.522107	0.140230	0.778982

```
[36]: viz_measures_model("Conv 1D", conv1D_measures)
```

Visualizing Conv 1D Performance in Each Fold

	tp	tn	fp	fn	tpr	tnr	precision	npv	fpr	\
1	58	169	194	15	0.794521	0.465565	0.230159	0.918478	0.534435	
2	55	246	113	22	0.714286	0.685237	0.327381	0.917910	0.314763	
3	66	124	237	8	0.891892	0.343490	0.217822	0.939394	0.656510	
4	66	263	86	20	0.767442	0.753582	0.434211	0.929329	0.246418	
5	46	302	69	18	0.718750	0.814016	0.400000	0.943750	0.185984	
6	62	258	101	14	0.815789	0.718663	0.380368	0.948529	0.281337	
7	71	258	93	13	0.845238	0.735043	0.432927	0.952030	0.264957	
8	82	199	145	9	0.901099	0.578488	0.361233	0.956731	0.421512	
9	77	185	164	9	0.895349	0.530086	0.319502	0.953608	0.469914	
10	48	286	79	22	0.685714	0.783562	0.377953	0.928571	0.216438	

	fdr	fnr	acc	f1	err_rate	bacc	tss	\
1	0.769841	0.205479	0.520642	0.356923	0.479358	0.630043	0.260085	
2	0.672619	0.285714	0.690367	0.448980	0.309633	0.699761	0.399522	
3	0.782178	0.108108	0.436782	0.350133	0.563218	0.617691	0.235382	
4	0.565789	0.232558	0.756322	0.554622	0.243678	0.760512	0.521024	
5	0.600000	0.281250	0.800000	0.513966	0.200000	0.766383	0.532766	
6	0.619632	0.184211	0.735632	0.518828	0.264368	0.767226	0.534452	
7	0.567073	0.154762	0.756322	0.572581	0.243678	0.790140	0.580281	
8	0.638767	0.098901	0.645977	0.515723	0.354023	0.739794	0.479587	
9	0.680498	0.104651	0.602299	0.470948	0.397701	0.712717	0.425435	
10	0.622047	0.314286	0.767816	0.487310	0.232184	0.734638	0.469276	

	hss	bs	auc
1	0.131391	0.251812	0.706253
2	0.272867	0.193838	0.782404
3	0.105547	0.282263	0.773040

4	0.404156	0.174732	0.822116
5	0.400665	0.158911	0.832800
6	0.368283	0.187712	0.848629
7	0.425978	0.176220	0.852395
8	0.309495	0.226460	0.825965
9	0.253371	0.254623	0.799227
10	0.353088	0.144878	0.864423

```
[26]: ## Average measures
def calc_avg_measures(measures):
    fpr_values, tpr_values = [], []
    avg = {}
    metrics = measures[0].keys();
    for metric in metrics:
        for i in range(0, 10):
            avg[metric] = avg.get(metric, 0) + measures[i][metric]
            if metric == 'fpr':
                fpr_values.append(measures[i][metric])
            elif metric == 'tpr':
                tpr_values.append(measures[i][metric])
        avg[metric] = avg[metric] / 10
    return avg

viz_measures_k_fold(0, RandomForest = calc_avg_measures(rf_measures),
    ↳DecisionTree=calc_avg_measures(dt_measures),
    ↳Conv1D=calc_avg_measures(conv1D_measures))
```

### Visualizing Model Performance

Measure	RandomForest	DecisionTree	Conv1D
tp	43.10	48.80	63.10
tn	349.30	323.60	229.00
fp	7.80	33.50	128.10
fn	35.00	29.30	15.00
tpr	0.55	0.63	0.80
tnr	0.98	0.91	0.64
precision	0.85	0.59	0.35
npv	0.91	0.92	0.94
fpr	0.02	0.09	0.36
fdr	0.15	0.41	0.65
fnr	0.45	0.37	0.20
acc	0.90	0.86	0.67
f1	0.67	0.61	0.48
err_rate	0.10	0.14	0.33
bacc	0.77	0.77	0.72
tss	0.53	0.53	0.44
hss	0.61	0.52	0.30

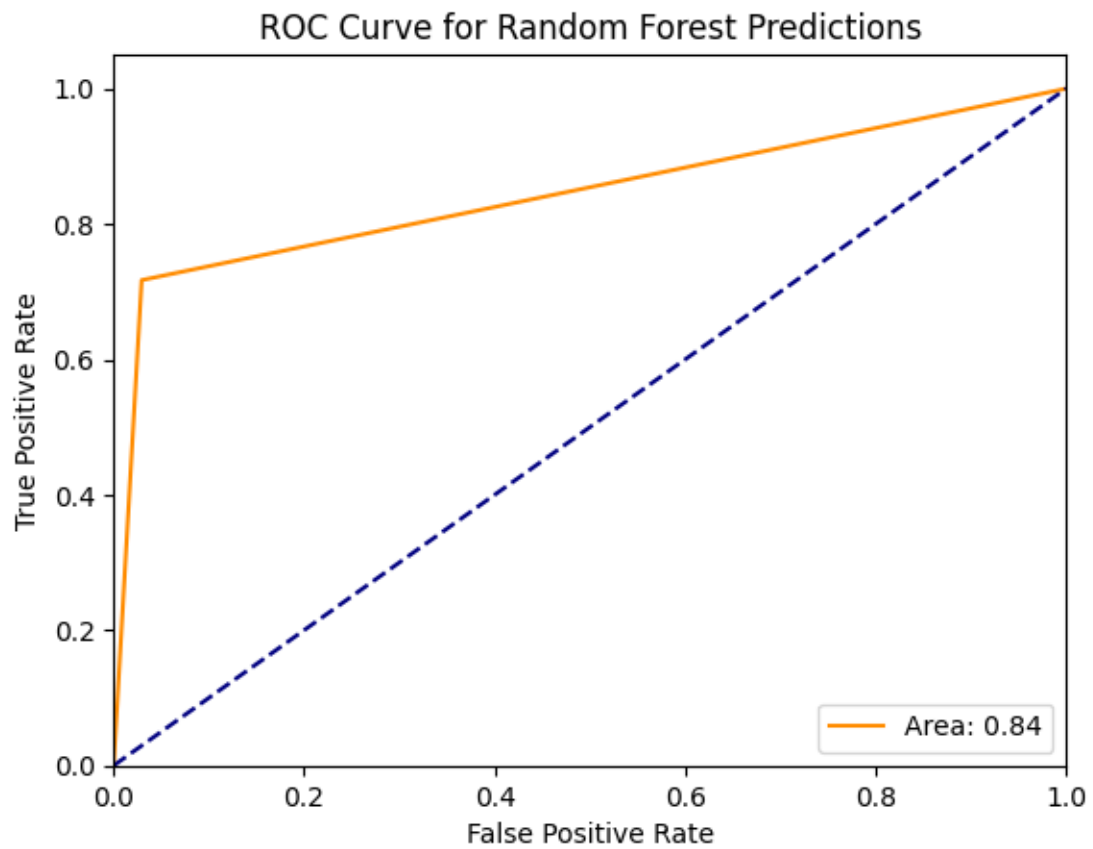
bs	0.07	0.14	0.21
auc	0.93	0.74	0.81

## 10 Visualizing ROC and Evaluating AUC of models using Test Datasets

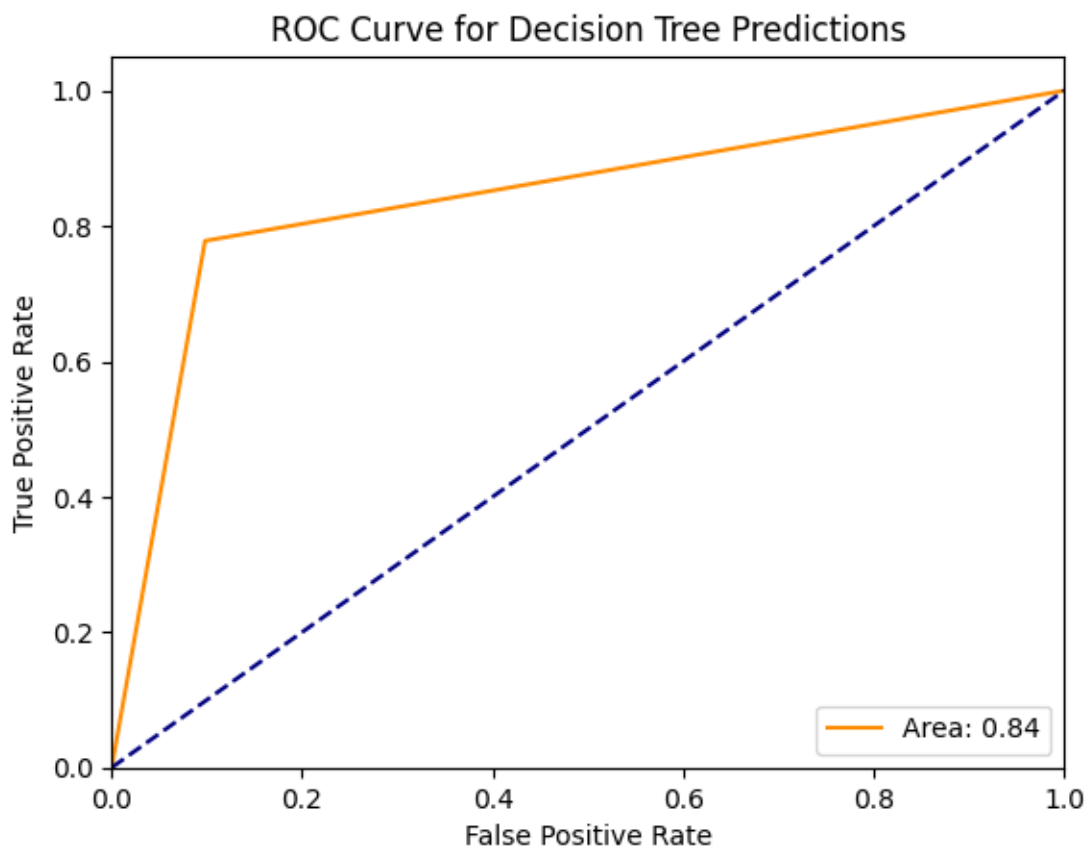
```
[27]: from sklearn.metrics import roc_curve, auc

def visualize_roc(y_test, y_pred, predictor):
    fpr, tpr, _ = roc_curve(y_test, y_pred)
    roc_auc = auc(fpr, tpr)
    plt.figure()
    plt.plot(fpr, tpr, color="darkorange", label=f'Area: {roc_auc:.2f}')
    plt.plot([0, 1], [0, 1], color="navy", linestyle="--")
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel("False Positive Rate")
    plt.ylabel("True Positive Rate")
    plt.title(f'ROC Curve for {predictor} Predictions')
    plt.legend(loc="lower right")
    plt.show()

[28]: y_pred = rf.predict(data_X_test)
visualize_roc(data_y_test, y_pred, "Random Forest")
```



```
[29]: y_pred = dt.predict(data_X_test)
      visualize_roc(data_y_test, y_pred, "Decision Tree")
```

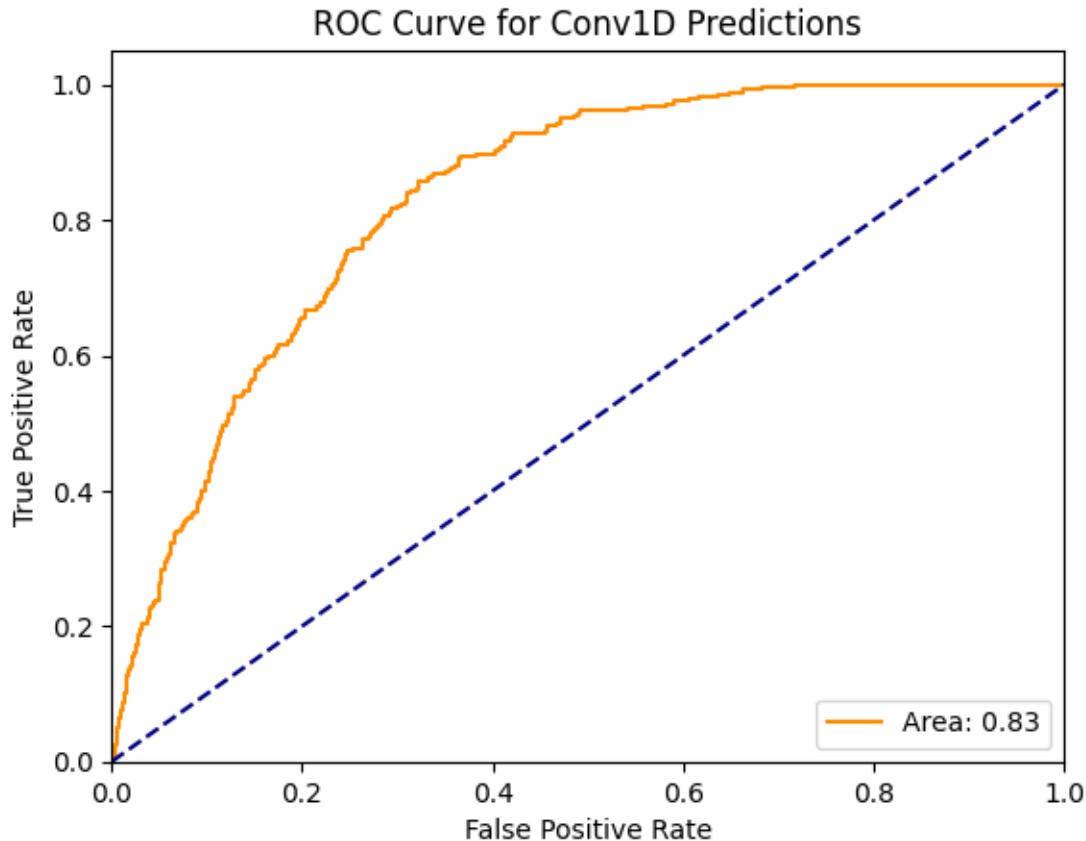


```
[30]: y_pred = model.predict(data_X_test)
      visualize_roc(data_y_test, y_pred, "Conv1D")
```

68/68

0s 199us/step





## 11 Evaluating Models

### 11.1 Analysing models by individual performance metric

Metrics like accuracy, FPR, and FNR can be misleading because our dataset is imbalanced. However, metrics such as F1, AUC, and balanced accuracy provide more insight into the model's performance, especially for the minority class. Multicollinearity can distort model coefficients and cause instability, potentially making precision, recall, and AUC less reliable and leading to higher error rates. Multicollinearity usually results in less interpretable models, which may affect the reliability of all metrics, particularly those dependent on feature weights or coefficients.

- **False Positive (FP):** Random Forest had the lowest false positives (7.80), suggesting that it performs best in minimizing incorrect wine recommendations. This makes Random Forest preferable if minimizing false positives is a priority.
- **True Negative Rate (TNR):** Random Forest had the highest TNR (0.98), indicating high reliability in identifying wines that should not be recommended. This metric is essential as it demonstrates the model's strength in handling multicollinearity while accurately predicting negatives.
- **Negative Predictive Value (NPV):** Although Decision Tree has a high NPV, Random

Forest also performs well (0.91). Since class weights were adjusted, 1D ConvNet’s high NPV (0.94) could suggest it handles negative class predictions effectively in this dataset.

- **False Discovery Rate (FDR):** Random Forest’s FDR (0.15) is the lowest among the models, suggesting it has the fewest incorrect positive predictions relative to total positive predictions. This reinforces Random Forest as a robust choice for minimizing false discoveries.
- **F1-Score:** Random Forest had an F1-score of 0.67, which is better than Decision Tree (0.61) and Conv1D (0.48). The higher F1-score indicates that Random Forest maintains a good balance between precision and recall, which is beneficial for imbalanced datasets.
- **Error Rate (Err Rate):** Random Forest had the lowest error rate (0.10), indicating fewer overall prediction errors compared to Decision Tree (0.14) and Conv1D (0.33). This further supports Random Forest’s effectiveness in this dataset.
- **Balanced Accuracy (BACC):** Both Random Forest and Decision Tree have similar balanced accuracy scores (0.77), showing they manage the trade-off between sensitivity and specificity. Although 1D ConvNet has a lower BACC, its performance could still be improved with further tuning.
- **True Skill Statistic (TSS) and Heidke Skill Score (HSS):** Random Forest has slightly higher TSS (0.53) and HSS (0.62), indicating that it better captures the model’s performance in correctly identifying positive and negative instances.
- **Brier Score (BS):** Random Forest has the lowest Brier Score (0.07), which suggests that its predicted probabilities are closest to the actual outcomes.
- **Area Under the Curve (AUC):** Random Forest has the highest AUC (0.93), demonstrating the best ability to distinguish between positive and negative classes. A high AUC is particularly valuable in this imbalanced dataset, as it indicates robust discriminatory power.

## 11.2 Best model for the dataset

Based on these evaluations, Random Forest emerges as the best model for this dataset due to its overall superior performance across critical metrics, particularly in handling false positives, maintaining high true negative rate, and achieving high AUC and balanced accuracy. Random Forest’s resilience to multicollinearity further solidifies its reliability.

However, if interpretability is essential, Decision Tree may offer some advantages due to its simpler structure, despite lower performance metrics. Meanwhile, Conv1D could potentially be improved with further tuning but currently shows less favorable results for this dataset.

## 12 Appendix

- Data Source:
  - <https://archive.ics.uci.edu/dataset/186/wine+quality>
- Packages used:
  - [https://pandas.pydata.org/docs/user\\_guide/index.html](https://pandas.pydata.org/docs/user_guide/index.html)
  - <https://matplotlib.org/stable/users/index>

- <https://seaborn.pydata.org/tutorial.html>
- <https://numpy.org/doc/2.1/>
- [https://mypy.readthedocs.io/en/stable/typed\\_dict.html](https://mypy.readthedocs.io/en/stable/typed_dict.html)
- [https://scikit-learn.org/stable/user\\_guide.html](https://scikit-learn.org/stable/user_guide.html)
- <https://www.tensorflow.org/install>