Imports

In [1]:
```
# Standard libraries
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import warnings

# Preprocessing
from statsmodels.stats.outliers_influence import variance_inflation_factor
from sklearn.preprocessing import StandardScaler

# Data splitting and hyperparameter tuning
from sklearn.model_selection import train_test_split, GridSearchCV

# Classification models
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis, Quadra
from sklearn.tree import DecisionTreeClassifier, plot_tree, export_graphviz
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from xgboost import XGBClassifier

# Metrics and evaluation
from sklearn.metrics import (
    accuracy_score,
    confusion_matrix,
    classification_report
)

# Neural networks (Keras / TensorFlow)
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.neural_network import MLPClassifier
from tensorflow.keras import layers
from tensorflow import keras
import tensorflow as tf

# Tree visualization
import graphviz

# Suppress warnings
warnings.filterwarnings("ignore")
```

Data Prep

In [2]:
```
df = pd.read_csv("../data/classification.csv")
```

In [3]:
```
df = df.dropna()
```

In [4]:
```
X = df.drop("Y", axis=1)
y = df["Y"]
```

In [5]:
```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)
```

In [6]:
```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled  = scaler.transform(X_test)
```

confusion_matrix Function (Evaluación)

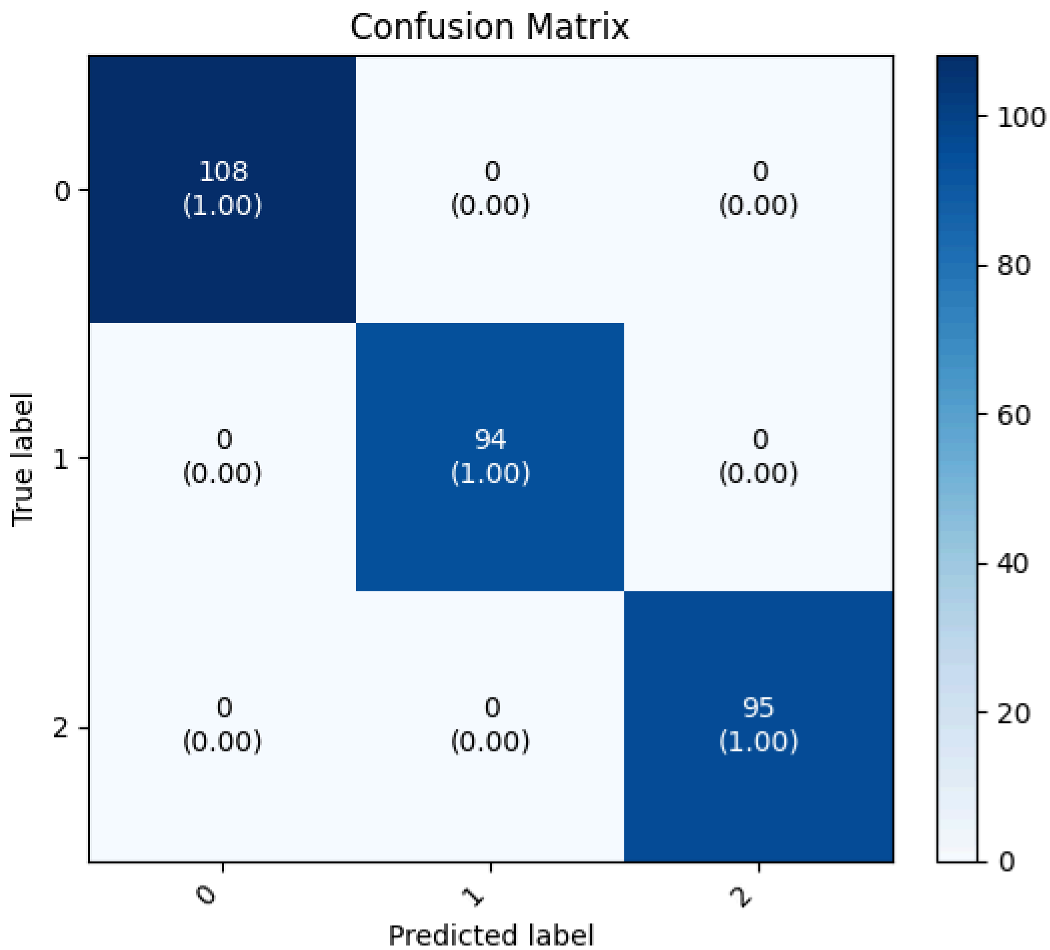
In [7]:
```
def plot_confusion_matrix(y_true, y_pred, classes=None, normalize=True,
                          title='Confusion Matrix', cmap=plt.cm.Blues, figsi
    """
    Plots a confusion matrix with counts y, opcionalmente, proporciones.
    """
    if classes is None:
        classes = np.unique(np.concatenate([y_true, y_pred]))

    cm = confusion_matrix(y_true, y_pred, labels=classes)
    cm = np.array(cm, dtype=int)

    if normalize:
        with np.errstate(all='ignore'):
            cm_norm = cm.astype('float') / cm.sum(axis=1, keepdims=True)
            cm_norm = np.nan_to_num(cm_norm, 0.0)
    else:
        cm_norm = np.zeros_like(cm, dtype=float)

    plt.figure(figsize=figsize)
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar(fraction=0.046, pad=0.04)

    ticks = np.arange(len(classes))
    plt.xticks(ticks, classes, rotation=45, ha='right')
    plt.yticks(ticks, classes)
    plt.xlabel('Predicted label')
    plt.ylabel('True label')

    thresh = cm.max() / 2
    for i in range(cm.shape[0]):
        for j in range(cm.shape[1]):
            count = cm[i, j]
            if normalize:
                prop = cm_norm[i, j]
                label = f"{count}\n({prop:.2f})"
            else:
                label = f"{count}"
            plt.text(j, i, label,
```

```
                    ha="center", va="center",
                    color="white" if count > thresh else "black")

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

Logistic Regression

In [8]:
```
logistic_model = LogisticRegression()

logistic_model.fit(X_train, y_train)

y_pred = logistic_model.predict(X_test)

print("Classification Report:\n", classification_report(y_test, y_pred))
plot_confusion_matrix(y_test, y_pred, classes=logistic_model.classes_)
```

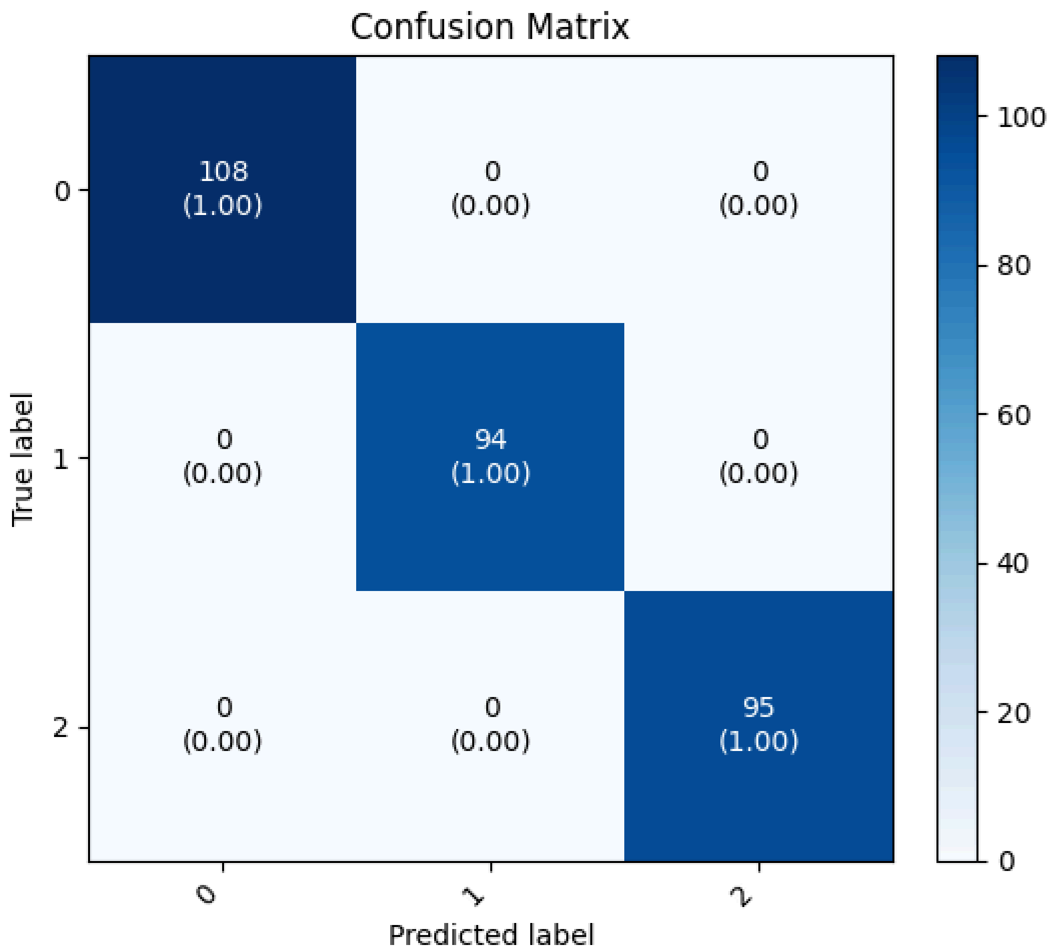
```
Classification Report:
               precision    recall  f1-score   support

           0       1.00      1.00      1.00       108
           1       1.00      1.00      1.00        94
           2       1.00      1.00      1.00        95

    accuracy                           1.00       297
   macro avg       1.00      1.00      1.00       297
weighted avg       1.00      1.00      1.00       297
```



LDA

In [9]:
```
lda_model = LinearDiscriminantAnalysis()
lda_model.fit(X_train_scaled, y_train)

y_pred = lda_model.predict(X_test_scaled)

print("Classification Report:\n", classification_report(y_test, y_pred))
plot_confusion_matrix(y_test, y_pred, classes=lda_model.classes_)
```

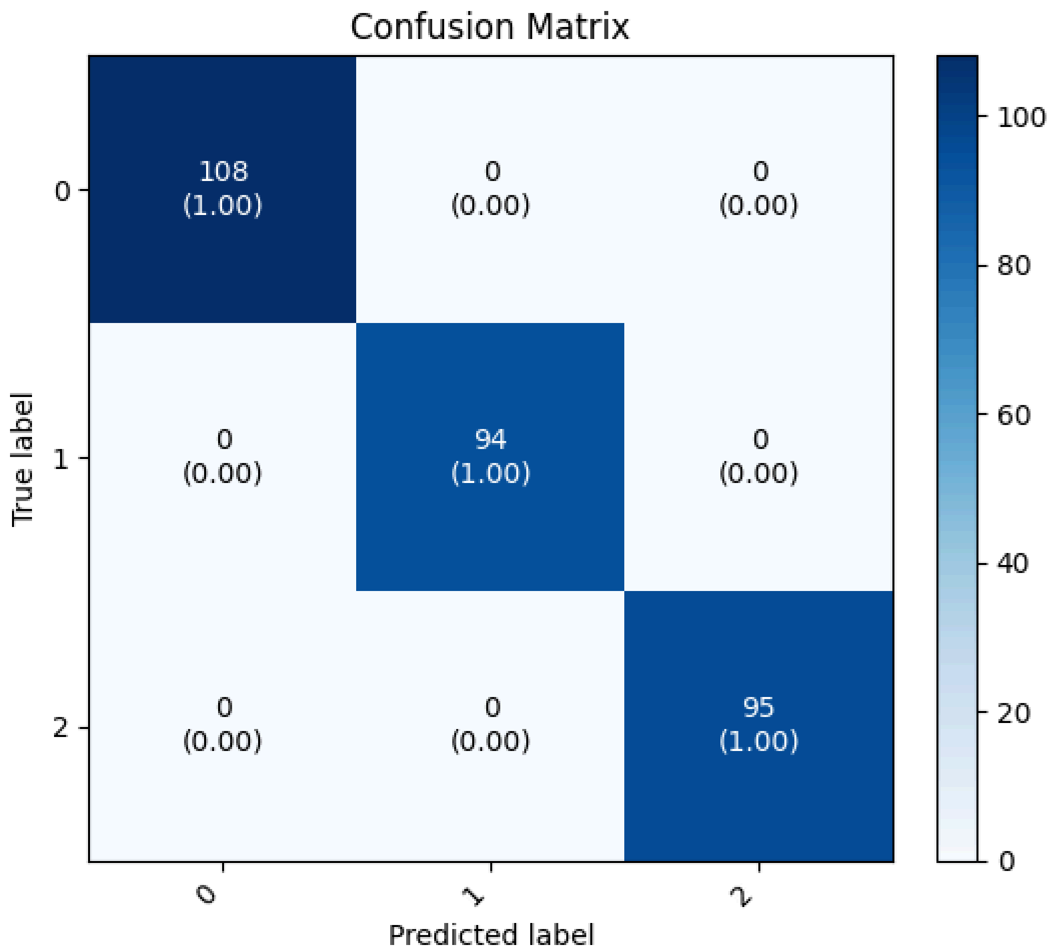
```
Classification Report:
               precision    recall  f1-score   support

           0       1.00      1.00      1.00       108
           1       1.00      1.00      1.00        94
           2       1.00      1.00      1.00        95

    accuracy                           1.00       297
   macro avg       1.00      1.00      1.00       297
weighted avg       1.00      1.00      1.00       297
```



QDA

In [10]:
```
qda_model = QuadraticDiscriminantAnalysis()
qda_model.fit(X_train_scaled, y_train)

y_pred = qda_model.predict(X_test_scaled)

print("Classification Report:\n", classification_report(y_test, y_pred))
plot_confusion_matrix(y_test, y_pred, classes=qda_model.classes_)
```

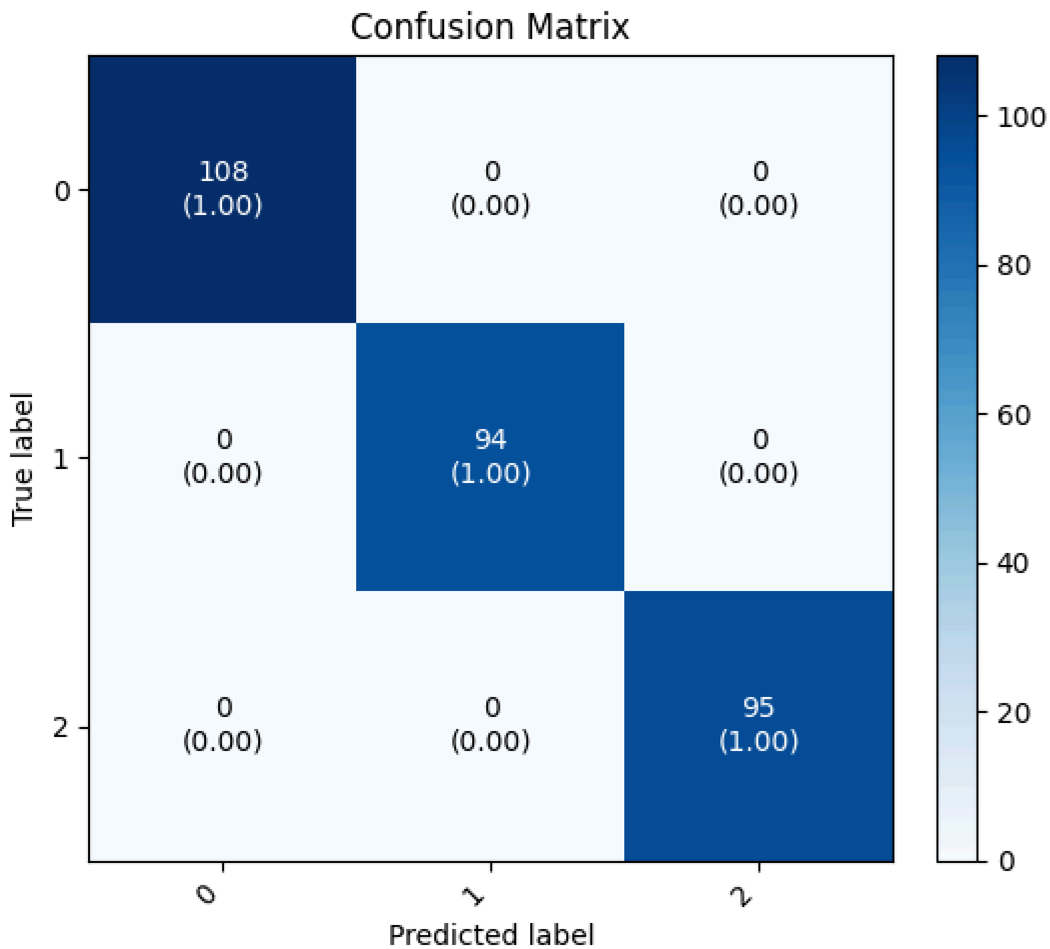
```
Classification Report:
               precision    recall  f1-score   support

           0       1.00      1.00      1.00       108
           1       1.00      1.00      1.00        94
           2       1.00      1.00      1.00        95

    accuracy                           1.00       297
   macro avg       1.00      1.00      1.00       297
weighted avg       1.00      1.00      1.00       297
```



KNN

In [11]:
```
knn_model = KNeighborsClassifier(n_neighbors=5)
knn_model.fit(X_train_scaled, y_train)

y_pred = knn_model.predict(X_test_scaled)

print("Classification Report:\n", classification_report(y_test, y_pred))
plot_confusion_matrix(y_test, y_pred, classes=knn_model.classes_)
```

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

           0       1.00      1.00      1.00       108
           1       1.00      1.00      1.00        94
           2       1.00      1.00      1.00        95

    accuracy                           1.00       297
   macro avg       1.00      1.00      1.00       297
weighted avg       1.00      1.00      1.00       297
```



Multicolinealidad

In [12]:
```
X_features = df.drop("Y", axis=1)

vif_data = pd.DataFrame()
vif_data["feature"] = X_features.columns
vif_data["VIF"] = [
    variance_inflation_factor(X_features.values, i)
    for i in range(len(X_features.columns))
]

display(vif_data.sort_values(by="VIF", ascending=False))
```

	feature	VIF
6	X7	52.978855
13	X14	42.144400
10	X11	41.025384
1	X2	39.682182
12	X13	38.514948
9	X10	35.837102
8	X9	34.093331
5	X6	30.431179
14	X15	29.243941
11	X12	25.662399
2	X3	23.687716
0	X1	16.243252
4	X5	16.189734
3	X4	12.689104
7	X8	7.582129

In [13]:
```
X_features = df.drop("Y", axis=1).copy()
threshold = 5

while True:
    vif_data = pd.DataFrame(
        {
            "feature": X_features.columns,
            "VIF": [
                variance_inflation_factor(X_features.values, i)
                for i in range(X_features.shape[1])
            ],
        }
    ).sort_values("VIF", ascending=False)

    max_vif = vif_data["VIF"].iloc[0]
    drop_feature = vif_data["feature"].iloc[0]
    print(vif_data, "\n")

    if max_vif <= threshold:
        print("All VIF values are below the threshold!")
        break

    if drop_feature in X_features.columns:
        print(f"Dropping {drop_feature} with VIF = {max_vif:.2f}")
        X_features = X_features.drop(columns=[drop_feature])
    else:
```

```
        print(f"» Warning: {drop_feature} not found in X_features, skipping.
        break
```

```
   feature        VIF
6       X7  52.978855
13     X14  42.144400
10     X11  41.025384
1       X2  39.682182
12     X13  38.514948
9      X10  35.837102
8       X9  34.093331
5       X6  30.431179
14     X15  29.243941
11     X12  25.662399
2       X3  23.687716
0       X1  16.243252
4       X5  16.189734
3       X4  12.689104
7       X8   7.582129

Dropping X7 with VIF = 52.98
   feature        VIF
12     X14  39.303975
1       X2  38.754066
9      X11  37.517855
11     X13  36.587792
8      X10  35.708046
7       X9  33.336703
13     X15  29.206697
5       X6  28.931070
10     X12  25.416008
2       X3  23.643420
4       X5  16.186650
0       X1  16.155865
3       X4  12.687135
6       X8   7.562611

Dropping X14 with VIF = 39.30
   feature        VIF
1       X2  38.646431
11     X13  35.944231
9      X11  34.497786
8      X10  33.931380
7       X9  32.343365
12     X15  29.200065
5       X6  27.481546
10     X12  24.910346
2       X3  23.460596
4       X5  16.026280
0       X1  15.909752
3       X4  12.472818
6       X8   7.535713

Dropping X2 with VIF = 38.65
   feature        VIF
8      X11  34.303007
7      X10  33.312464
6       X9  30.438274
11     X15  28.682611
```

```
4       X6  27.329226
10     X13  26.667239
9      X12  24.841094
1       X3  20.004341
0       X1  15.837771
3       X5  14.287058
2       X4  12.472812
5       X8   7.462880

Dropping X11 with VIF = 34.30
   feature        VIF
7      X10  32.538044
6       X9  28.418942
10     X15  27.377198
9      X13  24.926581
8      X12  24.765253
4       X6  22.058579
1       X3  19.627985
0       X1  15.836812
3       X5  13.801904
2       X4  12.447067
5       X8   7.191821

Dropping X10 with VIF = 32.54
  feature        VIF
6      X9  27.207333
8     X13  24.613787
9     X15  24.077533
7     X12  23.400594
4      X6  22.055865
1      X3  17.627564
0      X1  15.533186
3      X5  13.045315
2      X4  11.903332
5      X8   7.185832

Dropping X9 with VIF = 27.21
  feature        VIF
8     X15  24.045935
6     X12  21.338120
4      X6  19.765313
7     X13  17.913532
1      X3  17.602452
0      X1  14.739565
3      X5  13.034678
2      X4  11.645373
5      X8   7.139719

Dropping X15 with VIF = 24.05
  feature        VIF
6     X12  18.046648
7     X13  17.874209
1      X3  17.577368
4      X6  13.423457
0      X1  13.309943
3      X5  12.828299
```

```
2      X4  11.149767
5      X8   6.850980

Dropping X12 with VIF = 18.05
  feature        VIF
1      X3  16.749776
6     X13  15.953223
4      X6  12.859313
3      X5  12.775819
2      X4   9.380595
0      X1   9.165368
5      X8   6.842305

Dropping X3 with VIF = 16.75
  feature        VIF
3      X6  12.824892
5     X13  11.490511
2      X5   9.375834
0      X1   8.619086
1      X4   7.989880
4      X8   6.644837

Dropping X6 with VIF = 12.82
  feature       VIF
1      X4  7.897366
0      X1  7.618346
2      X5  6.158065
4     X13  4.987985
3      X8  4.633103

Dropping X4 with VIF = 7.90
  feature       VIF
1      X5  5.405320
3     X13  4.987641
2      X8  4.609960
0      X1  2.151998

Dropping X5 with VIF = 5.41
  feature       VIF
2     X13  1.294567
1      X8  1.169073
0      X1  1.120698

All VIF values are below the threshold!
```

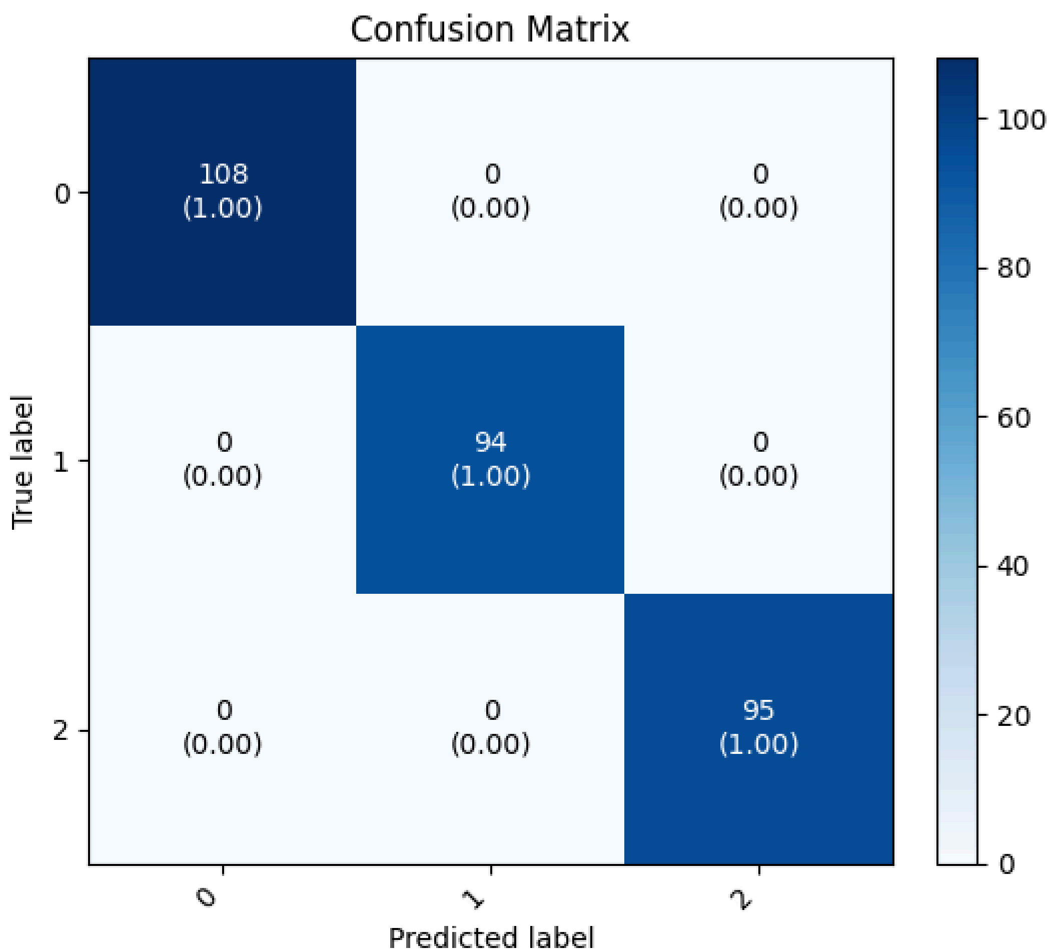
In [14]:
```python
print(f"Final number of features: {X_features.shape[1]}")
print("Remaining features:", list(X_features.columns))
```

```
Final number of features: 3
Remaining features: ['X1', 'X8', 'X13']
```

In [15]:
```python
selected_features = X_features.columns
X_selected = X_features[selected_features]
y_selected = df["Y"]

X_train_vif, X_test_vif, y_train_vif, y_test_vif = train_test_split(
```

```
    X_selected, y_selected, test_size=0.2, random_state=42
)

logistic_model = LogisticRegression()
logistic_model.fit(X_train_vif, y_train_vif)

y_pred = logistic_model.predict(X_test_vif)

print("Classification Report:\n", classification_report(y_test_vif, y_pred))
plot_confusion_matrix(y_test_vif, y_pred, classes=logistic_model.classes_)
```

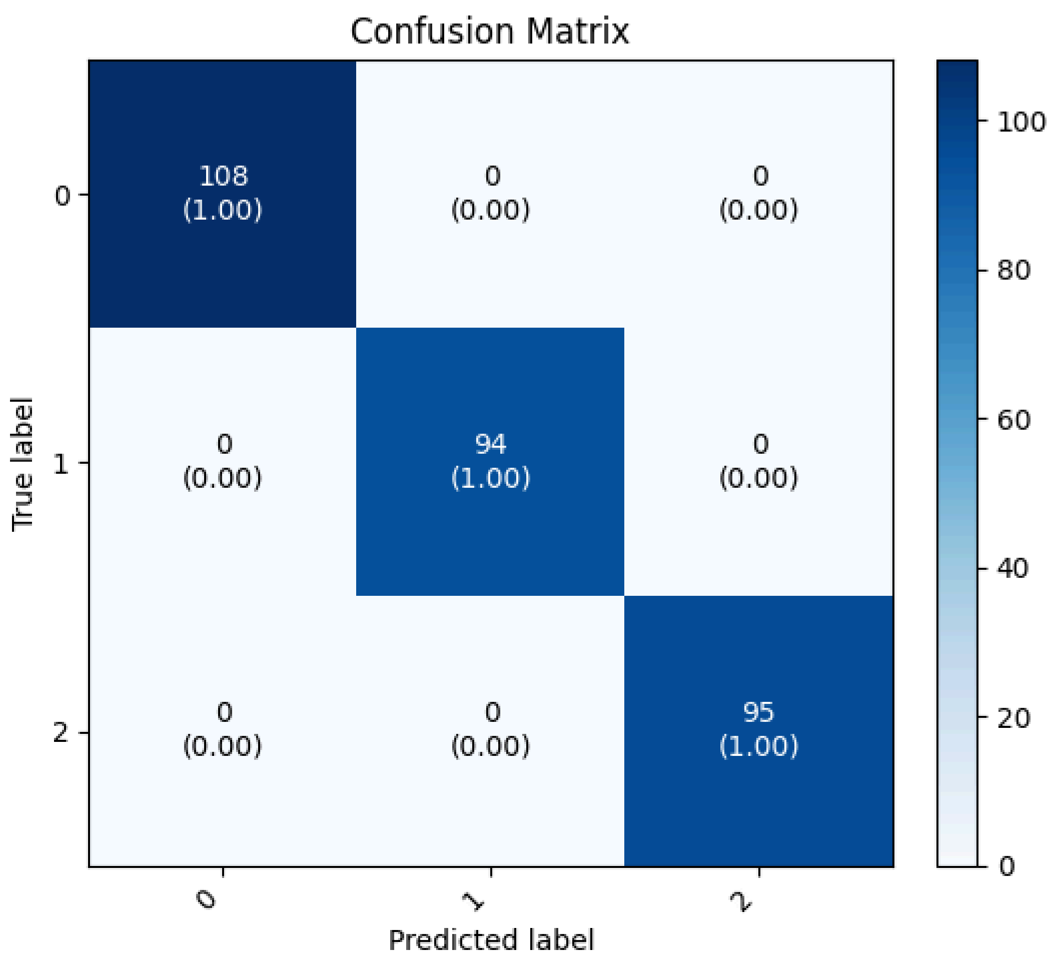
```
Classification Report:
               precision    recall  f1-score   support

           0       1.00      1.00      1.00       108
           1       1.00      1.00      1.00        94
           2       1.00      1.00      1.00        95

    accuracy                           1.00       297
   macro avg       1.00      1.00      1.00       297
weighted avg       1.00      1.00      1.00       297
```



DecisionTree

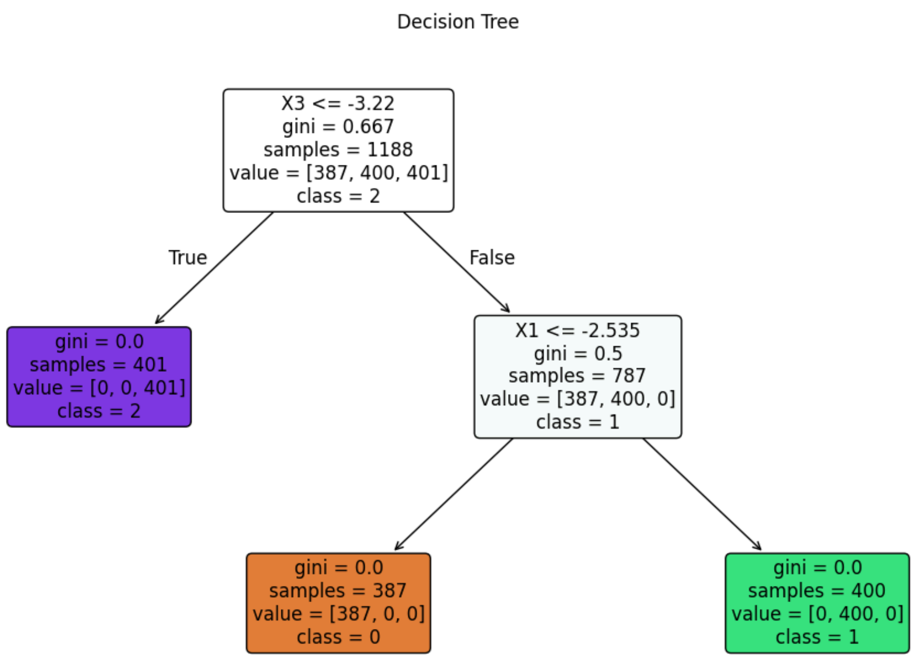
In [16]:
```
decisiontree_model = DecisionTreeClassifier(random_state=42)
decisiontree_model.fit(X_train_vif, y_train_vif)


y_pred = decisiontree_model.predict(X_test_vif)

print("Classification Report:\n", classification_report(y_test, y_pred))
plot_confusion_matrix(y_test, y_pred, classes=decisiontree_model.classes_)
```

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

           0       1.00      1.00      1.00       108
           1       1.00      1.00      1.00        94
           2       1.00      1.00      1.00        95

    accuracy                           1.00       297
   macro avg       1.00      1.00      1.00       297
weighted avg       1.00      1.00      1.00       297
```



In [17]:
```
dot_data = export_graphviz(
    decisiontree_model,
    out_file=None,
    feature_names=X_train_vif.columns,
    class_names=[str(c) for c in decisiontree_model.classes_],
    filled=True,
```

```
    rounded=True,
    special_characters=True,
)

graph = graphviz.Source(dot_data)
graph.render("tree", format="png", cleanup=False)
graph.view()

plt.figure(figsize=(14, 8))
plot_tree(
    decisiontree_model,
    filled=True,
    rounded=True,
    feature_names=X_train.columns,
    class_names=[str(c) for c in decisiontree_model.classes_],
    fontsize=12,
)
plt.title("Decision Tree")
plt.show()
```

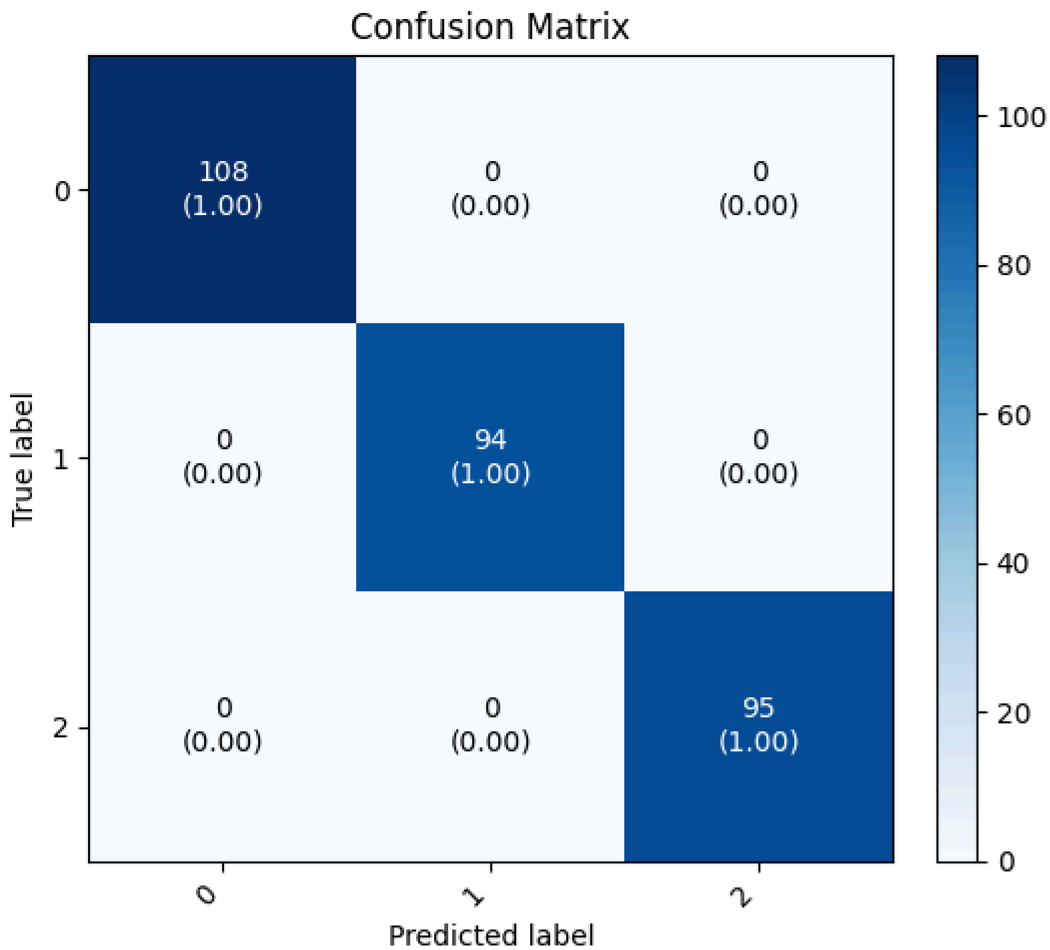


RandomForest

In [18]:
```
rf = RandomForestClassifier(random_state=42)

param_grid = {
    "n_estimators": [5, 10, 20],
    "max_depth": [None, 5, 10],
    "min_samples_split": [2, 5],
    "min_samples_leaf": [1, 2],
    "bootstrap": [True, False],
}
```

```
grid_search = GridSearchCV(
    estimator=rf, param_grid=param_grid, cv=5, scoring="accuracy", n_jobs=-1
)

grid_search.fit(X_train_scaled, y_train)

best_rf = grid_search.best_estimator_

y_pred = best_rf.predict(X_test_scaled)

print("Best parameters found:", grid_search.best_params_)
print("Classification Report:\n", classification_report(y_test, y_pred))
plot_confusion_matrix(y_test, y_pred, classes=best_rf.classes_)
```

```
Fitting 5 folds for each of 72 candidates, totalling 360 fits
Best parameters found: {'bootstrap': True, 'max_depth': None, 'min_samples_l
eaf': 1, 'min_samples_split': 2, 'n_estimators': 5}
Classification Report:
               precision    recall  f1-score   support

           0       1.00      1.00      1.00       108
           1       1.00      1.00      1.00        94
           2       1.00      1.00      1.00        95

    accuracy                           1.00       297
   macro avg       1.00      1.00      1.00       297
weighted avg       1.00      1.00      1.00       297
```

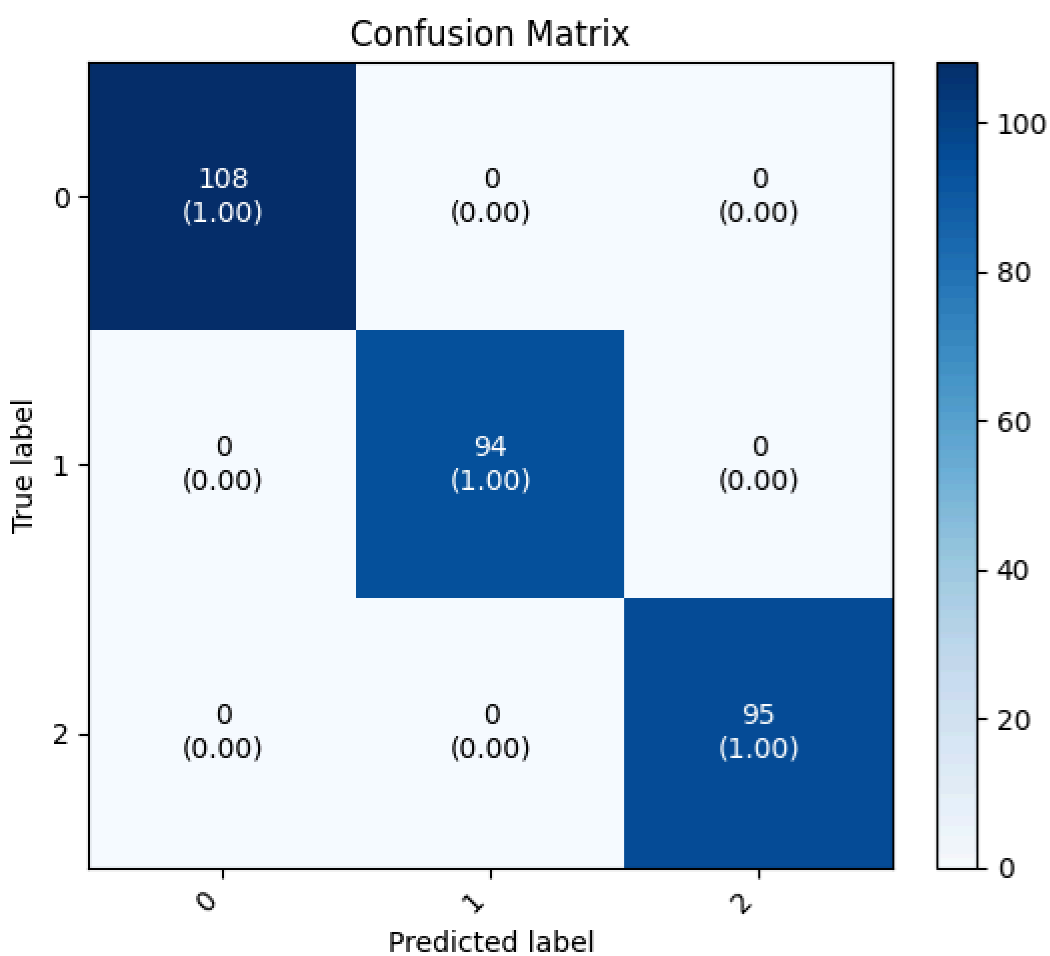


AdaBoost

In [19]:
```
base_estimator = DecisionTreeClassifier(random_state=42)

ada = AdaBoostClassifier(estimator=base_estimator, random_state=42)

param_grid_ada = {
    "n_estimators": [5, 10, 15],
    "learning_rate": [0.01, 0.1, 1.0],
    "estimator__max_depth": [1, 3, 5],
    "estimator__min_samples_split": [2, 5],
}

grid_search_ada = GridSearchCV(
    estimator=ada,
    param_grid=param_grid_ada,
    cv=5,
    scoring="accuracy",
    n_jobs=-1,
    verbose=1,
)

grid_search_ada.fit(X_train_scaled, y_train)
```

```python
best_ada = grid_search_ada.best_estimator_

y_pred = best_ada.predict(X_test_scaled)

print("Best parameters found:", grid_search_ada.best_params_)
print("Classification Report:\n", classification_report(y_test, y_pred))
plot_confusion_matrix(y_test, y_pred, classes=best_ada.classes_)
```

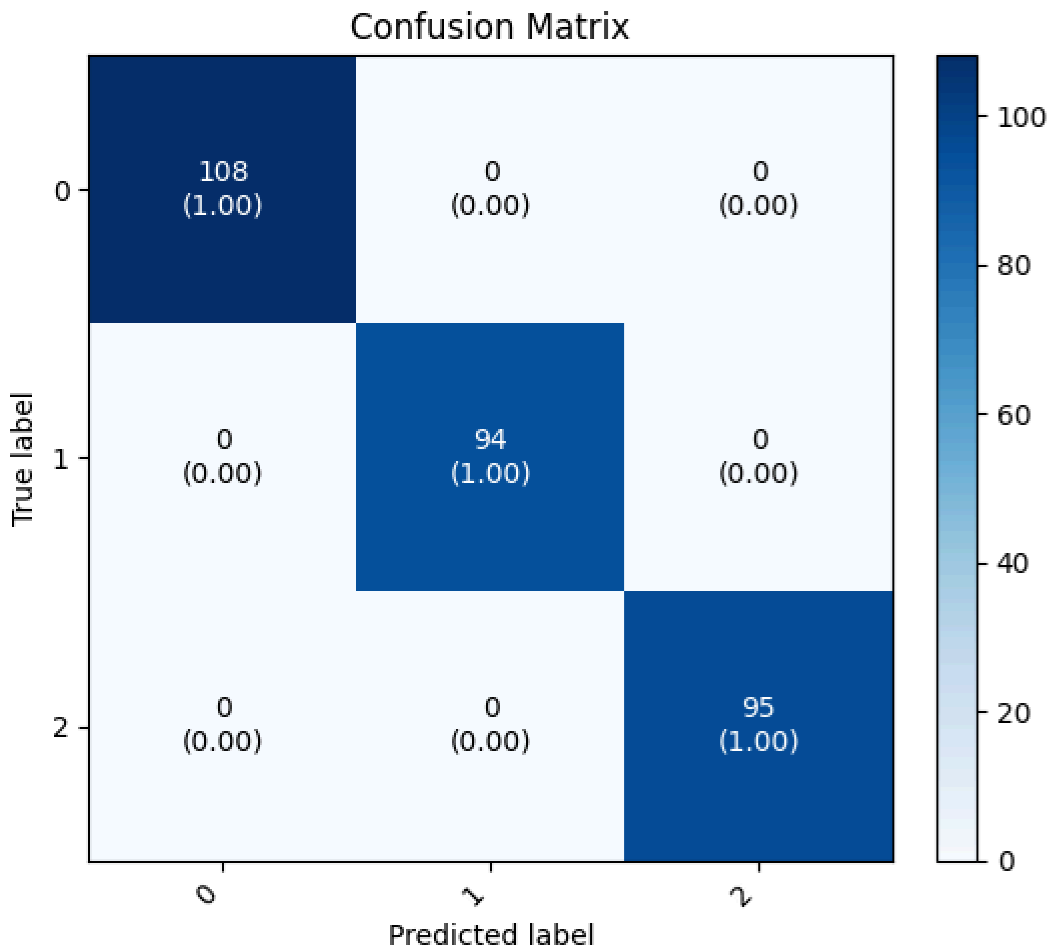
```
Fitting 5 folds for each of 54 candidates, totalling 270 fits
Best parameters found: {'estimator__max_depth': 1, 'estimator__min_samples_s
plit': 2, 'learning_rate': 0.1, 'n_estimators': 10}
Classification Report:
               precision    recall  f1-score   support

           0       1.00      1.00      1.00       108
           1       1.00      1.00      1.00        94
           2       1.00      1.00      1.00        95

    accuracy                           1.00       297
   macro avg       1.00      1.00      1.00       297
weighted avg       1.00      1.00      1.00       297
```



XGBClassifier

In [20]:
```
xgb = XGBClassifier(eval_metric="mlogloss", random_state=42)

param_grid_xgb = {
    "n_estimators": [5, 10, 15],
    "max_depth": [3, 5, 7],
    "learning_rate": [0.01, 0.1, 0.2],
    "subsample": [0.8, 1.0],
    "colsample_bytree": [0.8, 1.0],
}

grid_search_xgb = GridSearchCV(
    estimator=xgb,
    param_grid=param_grid_xgb,
    cv=5,
    scoring="accuracy",
    n_jobs=-1,
    verbose=1,
)

grid_search_xgb.fit(X_train_scaled, y_train)

best_xgb = grid_search_xgb.best_estimator_

y_pred = best_xgb.predict(X_test_scaled)

print("Best parameters found:", grid_search_xgb.best_params_)
print("Classification Report:\n", classification_report(y_test, y_pred))
plot_confusion_matrix(y_test, y_pred, classes=best_xgb.classes_)
```

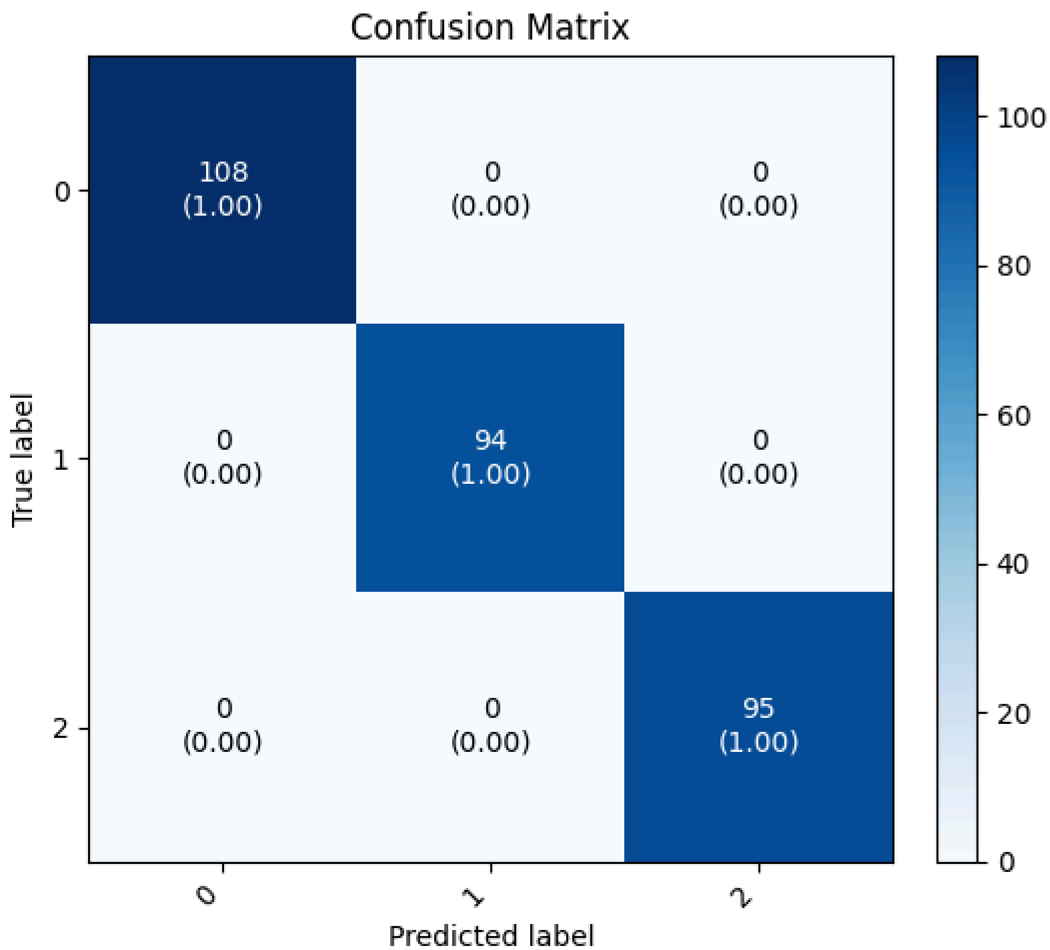
```
Fitting 5 folds for each of 108 candidates, totalling 540 fits
Best parameters found: {'colsample_bytree': 0.8, 'learning_rate': 0.01, 'max
_depth': 3, 'n_estimators': 5, 'subsample': 0.8}
Classification Report:
               precision    recall  f1-score   support

           0       1.00      1.00      1.00       108
           1       1.00      1.00      1.00        94
           2       1.00      1.00      1.00        95

    accuracy                           1.00       297
   macro avg       1.00      1.00      1.00       297
weighted avg       1.00      1.00      1.00       297
```



Redes Neuronales

In [21]:
```
clf = MLPClassifier(
    solver='lbfgs',
    alpha=1,
    hidden_layer_sizes=(60,),
    random_state=1,
    max_iter=1000
)

clf.fit(X_train_scaled, y_train)

y_pred = clf.predict(X_test_scaled)
accuracy = accuracy_score(y_test, y_pred)
print(f"\nMLPClassifier Accuracy: {accuracy:.4f}")

print("Classification Report:\n", classification_report(y_test, y_pred))
plot_confusion_matrix(y_test, y_pred, classes=clf.classes_)
```

```
MLPClassifier Accuracy: 1.0000
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       108
           1       1.00      1.00      1.00        94
           2       1.00      1.00      1.00        95

    accuracy                           1.00       297
   macro avg       1.00      1.00      1.00       297
weighted avg       1.00      1.00      1.00       297
```



In [22]:
```
def build_model():
    model = keras.Sequential([
        layers.Dense(64, activation='relu', input_shape=(X_train_scaled.shap
        layers.Dropout(0.3),
        layers.Dense(32, activation='relu'),
        layers.Dense(len(set(y_train)), activation='softmax')
    ])

    optimizer = tf.keras.optimizers.RMSprop(learning_rate=0.001)

    model.compile(
        loss='sparse_categorical_crossentropy',
        optimizer=optimizer,
        metrics=['accuracy']
```

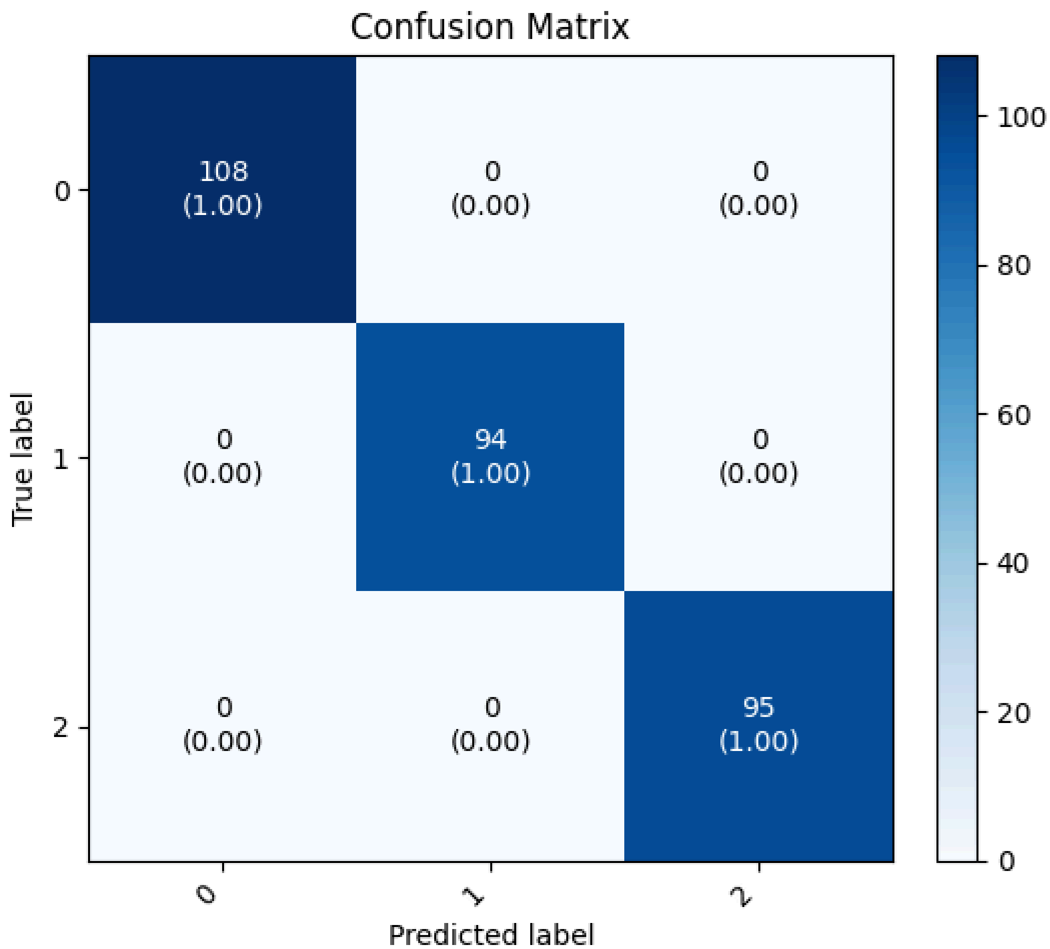
```
    )

    return model

early_stop = EarlyStopping(
    monitor='val_loss',
    patience=5,
    restore_best_weights=True,
    verbose=1
)

model_mlp = build_model()

history = model_mlp.fit(
    X_train_scaled,
    y_train,
    epochs=100,
    batch_size=16,
    validation_split=0.2,
    callbacks=[early_stop],
    verbose=1
)

loss_mlp, acc_mlp = model_mlp.evaluate(X_test_scaled, y_test, verbose=0)
print(f"\nMultilayer NN Accuracy: {acc_mlp:.4f}")
print("Classification Report:\n", classification_report(y_test, np.argmax(mo
plot_confusion_matrix(y_test, np.argmax(model_mlp.predict(X_test_scaled), ax
```

```
Epoch 1/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.8361 - loss: 0.5177 - v
al_accuracy: 1.0000 - val_loss: 0.0103
Epoch 2/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 1.0000 - loss: 0.0091 - v
al_accuracy: 1.0000 - val_loss: 2.8859e-04
Epoch 3/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 1.0000 - loss: 7.9190e-04
- val_accuracy: 1.0000 - val_loss: 2.7627e-05
Epoch 4/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 1.0000 - loss: 2.2098e-04
- val_accuracy: 1.0000 - val_loss: 7.1500e-06
Epoch 5/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 1.0000 - loss: 5.9339e-05
- val_accuracy: 1.0000 - val_loss: 3.8643e-06
Epoch 6/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 1.0000 - loss: 5.3627e-05
- val_accuracy: 1.0000 - val_loss: 2.5219e-06
Epoch 7/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 1.0000 - loss: 2.9222e-05
- val_accuracy: 1.0000 - val_loss: 1.8728e-06
Epoch 8/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 1.0000 - loss: 1.8189e-05
- val_accuracy: 1.0000 - val_loss: 1.4215e-06
Epoch 9/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 1.0000 - loss: 2.9366e-05
- val_accuracy: 1.0000 - val_loss: 1.0428e-06
Epoch 10/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 1.0000 - loss: 3.9214e-05
- val_accuracy: 1.0000 - val_loss: 7.3779e-07
Epoch 11/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 933us/step - accuracy: 1.0000 - loss: 1.5110e-
05 - val_accuracy: 1.0000 - val_loss: 6.1358e-07
Epoch 12/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 980us/step - accuracy: 1.0000 - loss: 2.4346e-
05 - val_accuracy: 1.0000 - val_loss: 4.7533e-07
Epoch 13/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 1.0000 - loss: 1.1442e-05
- val_accuracy: 1.0000 - val_loss: 4.1172e-07
Epoch 14/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 1.0000 - loss: 1.2709e-04
- val_accuracy: 1.0000 - val_loss: 3.4460e-07
Epoch 15/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 966us/step - accuracy: 1.0000 - loss: 1.6306e-
05 - val_accuracy: 1.0000 - val_loss: 2.9352e-07
Epoch 16/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 956us/step - accuracy: 1.0000 - loss: 7.9631e-
06 - val_accuracy: 1.0000 - val_loss: 2.6597e-07
Epoch 17/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 940us/step - accuracy: 1.0000 - loss: 5.9990e-
06 - val_accuracy: 1.0000 - val_loss: 2.4593e-07
Epoch 18/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 916us/step - accuracy: 1.0000 - loss: 1.0853e-
05 - val_accuracy: 1.0000 - val_loss: 2.0686e-07
Epoch 19/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 998us/step - accuracy: 1.0000 - loss: 1.2397e-
```

```
05 - val_accuracy: 1.0000 - val_loss: 1.6830e-07
Epoch 20/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 1.0000 - loss: 7.1196e-06
- val_accuracy: 1.0000 - val_loss: 1.5227e-07
Epoch 21/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 1.0000 - loss: 5.2040e-06
- val_accuracy: 1.0000 - val_loss: 1.2722e-07
Epoch 22/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 1.0000 - loss: 9.5500e-06
- val_accuracy: 1.0000 - val_loss: 1.0719e-07
Epoch 23/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 1.0000 - loss: 4.7473e-06
- val_accuracy: 1.0000 - val_loss: 1.0068e-07
Epoch 24/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 1.0000 - loss: 6.4098e-06
- val_accuracy: 1.0000 - val_loss: 9.7171e-08
Epoch 25/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 971us/step - accuracy: 1.0000 - loss: 2.8817e-
06 - val_accuracy: 1.0000 - val_loss: 9.3164e-08
Epoch 26/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 930us/step - accuracy: 1.0000 - loss: 3.4063e-
06 - val_accuracy: 1.0000 - val_loss: 8.5650e-08
Epoch 27/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 1.0000 - loss: 1.3360e-05
- val_accuracy: 1.0000 - val_loss: 7.7636e-08
Epoch 28/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 978us/step - accuracy: 1.0000 - loss: 2.5578e-
05 - val_accuracy: 1.0000 - val_loss: 6.5114e-08
Epoch 29/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 1.0000 - loss: 6.6706e-06
- val_accuracy: 1.0000 - val_loss: 5.9104e-08
Epoch 30/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 985us/step - accuracy: 1.0000 - loss: 4.9101e-
06 - val_accuracy: 1.0000 - val_loss: 5.4596e-08
Epoch 31/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 929us/step - accuracy: 1.0000 - loss: 2.1535e-
06 - val_accuracy: 1.0000 - val_loss: 5.2592e-08
Epoch 32/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 897us/step - accuracy: 1.0000 - loss: 4.5237e-
06 - val_accuracy: 1.0000 - val_loss: 4.8084e-08
Epoch 33/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 977us/step - accuracy: 1.0000 - loss: 5.8296e-
06 - val_accuracy: 1.0000 - val_loss: 4.2575e-08
Epoch 34/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 1.0000 - loss: 3.0285e-06
- val_accuracy: 1.0000 - val_loss: 3.7566e-08
Epoch 35/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 998us/step - accuracy: 1.0000 - loss: 5.2727e-
06 - val_accuracy: 1.0000 - val_loss: 3.5062e-08
Epoch 36/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 964us/step - accuracy: 1.0000 - loss: 6.6615e-
06 - val_accuracy: 1.0000 - val_loss: 3.4060e-08
Epoch 37/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 990us/step - accuracy: 1.0000 - loss: 3.2104e-
06 - val_accuracy: 1.0000 - val_loss: 3.1555e-08
Epoch 38/100
```

```
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 942us/step - accuracy: 1.0000 - loss: 2.9814e-
06 - val_accuracy: 1.0000 - val_loss: 2.9552e-08
Epoch 39/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 849us/step - accuracy: 1.0000 - loss: 3.0836e-
06 - val_accuracy: 1.0000 - val_loss: 2.9051e-08
Epoch 40/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 1.0000 - loss: 4.5232e-06
- val_accuracy: 1.0000 - val_loss: 2.7047e-08
Epoch 41/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 909us/step - accuracy: 1.0000 - loss: 4.5040e-
06 - val_accuracy: 1.0000 - val_loss: 2.6046e-08
Epoch 42/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 917us/step - accuracy: 1.0000 - loss: 6.6679e-
06 - val_accuracy: 1.0000 - val_loss: 2.3040e-08
Epoch 43/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 916us/step - accuracy: 1.0000 - loss: 3.8018e-
06 - val_accuracy: 1.0000 - val_loss: 2.2540e-08
Epoch 44/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 892us/step - accuracy: 1.0000 - loss: 3.8858e-
06 - val_accuracy: 1.0000 - val_loss: 2.0536e-08
Epoch 45/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 944us/step - accuracy: 1.0000 - loss: 3.4998e-
06 - val_accuracy: 1.0000 - val_loss: 1.9534e-08
Epoch 46/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 986us/step - accuracy: 1.0000 - loss: 5.3637e-
06 - val_accuracy: 1.0000 - val_loss: 1.9534e-08
Epoch 47/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 980us/step - accuracy: 1.0000 - loss: 4.7071e-
06 - val_accuracy: 1.0000 - val_loss: 1.7030e-08
Epoch 48/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 1.0000 - loss: 2.8793e-06
- val_accuracy: 1.0000 - val_loss: 1.5527e-08
Epoch 49/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 1.0000 - loss: 1.5018e-06
- val_accuracy: 1.0000 - val_loss: 1.5026e-08
Epoch 50/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 952us/step - accuracy: 1.0000 - loss: 1.0192e-
06 - val_accuracy: 1.0000 - val_loss: 1.5026e-08
Epoch 51/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 980us/step - accuracy: 1.0000 - loss: 2.9696e-
06 - val_accuracy: 1.0000 - val_loss: 1.4525e-08
Epoch 52/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 991us/step - accuracy: 1.0000 - loss: 1.4080e-
06 - val_accuracy: 1.0000 - val_loss: 1.4025e-08
Epoch 53/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 1.0000 - loss: 1.4981e-06
- val_accuracy: 1.0000 - val_loss: 1.3524e-08
Epoch 54/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 1.0000 - loss: 1.4140e-06
- val_accuracy: 1.0000 - val_loss: 1.3023e-08
Epoch 55/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 990us/step - accuracy: 1.0000 - loss: 1.7292e-
06 - val_accuracy: 1.0000 - val_loss: 1.2021e-08
Epoch 56/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 1.0000 - loss: 1.5172e-06
- val_accuracy: 1.0000 - val_loss: 1.1019e-08
```

```
Epoch 57/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 972us/step - accuracy: 1.0000 - loss: 1.4582e-
05 - val_accuracy: 1.0000 - val_loss: 9.5167e-09
Epoch 58/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 993us/step - accuracy: 1.0000 - loss: 1.7822e-
06 - val_accuracy: 1.0000 - val_loss: 9.5167e-09
Epoch 59/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 978us/step - accuracy: 1.0000 - loss: 1.9408e-
06 - val_accuracy: 1.0000 - val_loss: 9.0158e-09
Epoch 60/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 999us/step - accuracy: 1.0000 - loss: 1.5033e-
06 - val_accuracy: 1.0000 - val_loss: 8.0141e-09
Epoch 61/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 1.0000 - loss: 1.5759e-06
- val_accuracy: 1.0000 - val_loss: 8.0141e-09
Epoch 62/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 1.0000 - loss: 8.5726e-07
- val_accuracy: 1.0000 - val_loss: 8.0141e-09
Epoch 63/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 1.0000 - loss: 1.0031e-06
- val_accuracy: 1.0000 - val_loss: 8.0141e-09
Epoch 64/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 1.0000 - loss: 5.4083e-06
- val_accuracy: 1.0000 - val_loss: 8.0141e-09
Epoch 65/100
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 1.0000 - loss: 1.7630e-06
- val_accuracy: 1.0000 - val_loss: 8.0141e-09
Epoch 65: early stopping
Restoring model weights from the end of the best epoch: 60.

Multilayer NN Accuracy: 1.0000
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       108
           1       1.00      1.00      1.00        94
           2       1.00      1.00      1.00        95

    accuracy                           1.00       297
   macro avg       1.00      1.00      1.00       297
weighted avg       1.00      1.00      1.00       297

10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step
```



Comentarios

- Se entrenaron y probaron 10 modelos: Regresión Logística, LDA, QDA, K-NN, Árbol de Decisión, Random Forest, AdaBoost, XGBoost, un MLP y una red neuronal densa.
- Cada modelo entrega accuracy, precision, recall y F1 de 1.00 en las tres clases; las matrices de confusión son totalmente diagonales. El resultado se aprecia desde la Regresión Logística hasta la red neuronal.
- Desde el EDA se pudo apreciar que existían tres clústeres muy bien separados en prácticamente todas las variables y, en particular, en el trío (X1, X8, X13); es por eso que incluso un clasificador lineal basta para separar las clases sin error.
- Se usó Factor de Inflación de Varianza (VIF) para detectar multicolinealidad, descartando 12 de las 15 variables originales. Las tres restantes (X1, X8 y X13) son suficientes para lograr un 100 % de acierto en Regresión Logística y Árbol de Decisión, lo que demuestra que aún con solo estas 3 variables el modelo sigue siendo igual de efectivo.

- No era necesario realizar GridSearch y ajustar hiperparámetros, ya que los modelos se desempeñaron bien con sus configuraciones predeterminadas. Sin embargo, se exploraron algunos hiperparámetros:
  - *Random Forest*: la búsqueda selecciona solo 5 árboles y profundidad ilimitada, con eso basta para 100 % de precisión.
  - *AdaBoost*: óptimo con 10 *stumps* (prof. 1) y *learning_rate* 0.1.
  - *XGBoost*: 5 árboles, profundidad 3 y *eta* 0.01 ya son suficientes para alcanzar ese mismo 100 % de precisión.

  El punto anterior confirma la simplicidad del conjunto de datos.

- Para las redes neuronales se entrenaron dos modelos:
  - MLP de 60 neuronas (solver lbfgs) entrena rápidamente y obtiene 1.0 de accuracy.
  - Red Keras 64-32 con drop-out 0.3 y early stopping detiene el entrenamiento en la época 84 con val_loss $\approx 1 \times 10^{-8}$.

  Un valor de pérdida tan bajo y el 100 % en validación y prueba es provocado por la separabilidad tan marcada de los datos.

- En la realidad, para este problema, bastaría un modelo interpretable y sencillo como Regresión Logística (o LDA) con las tres variables seleccionadas. Esto nos da transparencia, rapidez y el mismo desempeño que otros modelos más complejos.