Wine Quality Prediction

Individual Project By:

Lana Gilmore

Course: CISB 60 – ML and DL (Fall, 2024)

Problem Statement

- This project focuses on predicting wine quality using machine learning and deep learning techniques, exploring chemical properties of wine as features.

- **Keywords:** Wine quality prediction, classification, machine learning, deep learning

Methodology

1. Approach for Machine Learning and Deep Learning:

This project applies machine learning and deep learning techniques to predict wine quality based on its chemical properties. The machine learning approach uses a Support Vector Machine (SVM) model, which efficiently handles high-dimensional data and separates classes using a hyperplane. For deep learning, a neural network is implemented to capture complex relationships in the data, leveraging its ability to model non-linear patterns. Hyperparameter tuning is performed to optimize learning rates and batch sizes, and TensorBoard is integrated to visualize training metrics, providing insights into the optimization process. Both methods focus on improving prediction accuracy and generalization across quality categories.

2. Models Used in the Project:

- **Model 1:** Support Vector Machine (SVM) for classification tasks.

- **Model 2:** Neural Network for classification tasks.

- **Techniques Applied:**

  - **Hyperparameter Tuning:** Experiments with learning rates and batch sizes to optimize performance.

  - **TensorBoard Integration:** Visualizations to monitor training metrics and track progress.

Lab: Fraud Detection in Wine Dataset using SVM with Grid Search

In this lab, you will be working on a dataset that contains wine types and their quality, classified as either "Legit" or "Fraud." Your task is to perform classification using Support Vector Machines (SVM) with hyperparameter tuning. You will complete several tasks to clean, preprocess, and analyze the data. Finally, you'll evaluate the model's performance.

Goal:

To understand how to apply classification algorithms and perform model evaluation on imbalanced datasets.

Dataset: wine_fraud.csv

This dataset includes the following columns:

- **type**: The type of wine (red or white).
- **quality**: The classification of the wine (Legit or Fraud).

Exploratory Data Analysis (EDA)

- The dataset is checked for missing values and overall structure.
- Visualizations are used to explore feature distributions and the target variable.
- Correlations between features and the target are analyzed to understand their relationships.

Task 1: Load the Dataset and Display the First Few Rows

Instruction: Load the dataset `wine_fraud.csv` into a DataFrame using Pandas and display the first five rows of data.

> **Machine Learning Section:** This section begins the implementation of machine learning models.

In [1]:
```
#Import required Libraries
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.metrics import classification_report, confusion_matrix


# This section completed by Lana Gilmore
```

In [2]:
```
# Load the dataset
df = pd.read_csv("data/wine_fraud.csv")

#Display the first 5 rows of data
df.head()

# This section completed by Lana Gilmore
```

Out[2]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	al
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	0.68	
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	0.65	
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	0.58	
4	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	

In [3]:
```
# Load the dataset
df = pd.read_csv("data/wine_fraud.csv")

#Display the first 5 rows of data
df.head()

# This section completed by Lana Gilmore
```

Out[3]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	al
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	0.68	
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	0.65	
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	0.58	
4	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	

Task 2: Explore the Target Variable (quality)

Instruction: Check the unique values in the target column `quality` to understand the classification types.

Hint: Use `.unique()` to display the distinct values in the column.

In [4]:
```
# Check unique values in the 'quality' column
df['quality'].unique()

# This section completed by Lana Gilmore
```

Out[4]:
```
array(['Legit', 'Fraud'], dtype=object)
```

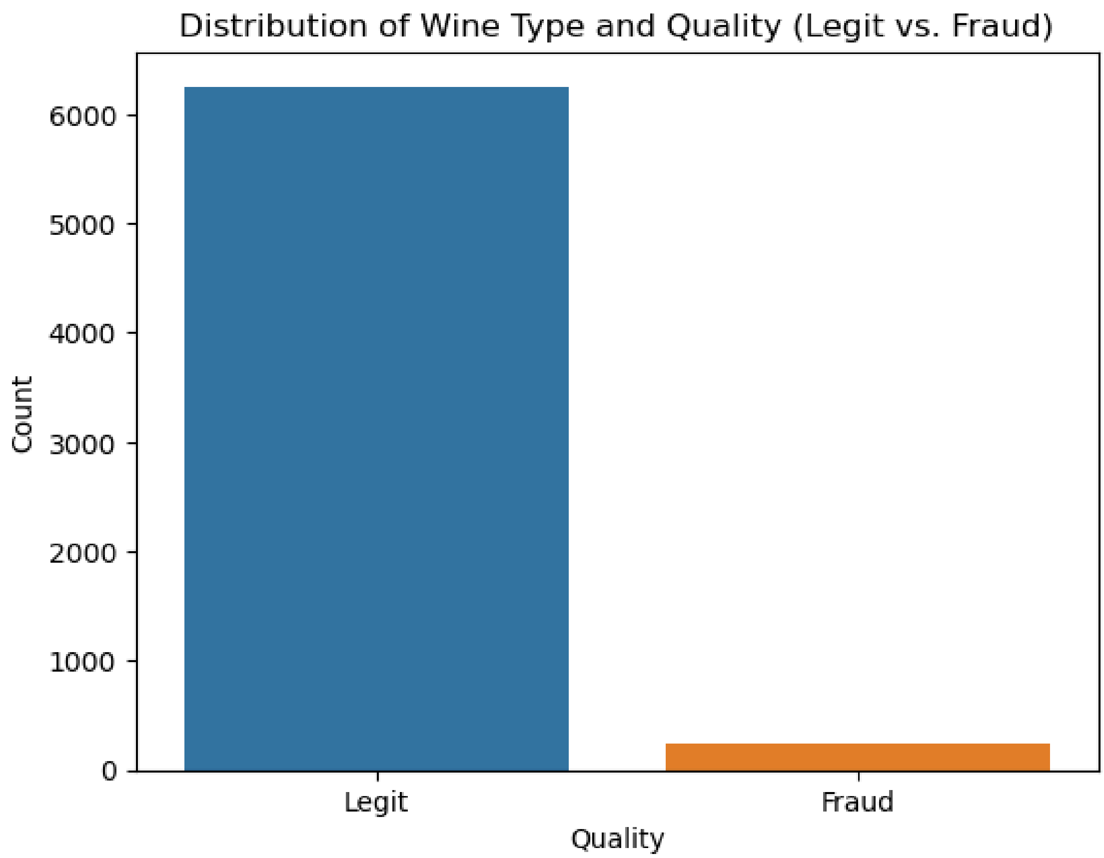
Task 3: Plot the Distribution of Legit vs. Fraud Wines

Instruction: Create a countplot to display the number of wines classified as "Legit" vs. "Fraud."

Hint: Use Seaborn’s `countplot()` function.

In [5]:
```
# Countplot to display number of wines classified as "Legit" vs. "Fraud"
sns.countplot(x='quality', data=df)
plt.title('Distribution of Wine Type and Quality (Legit vs. Fraud)')
plt.xlabel('Quality')
plt.ylabel('Count')
plt.show()

# This section completed by Lana Gilmore
```

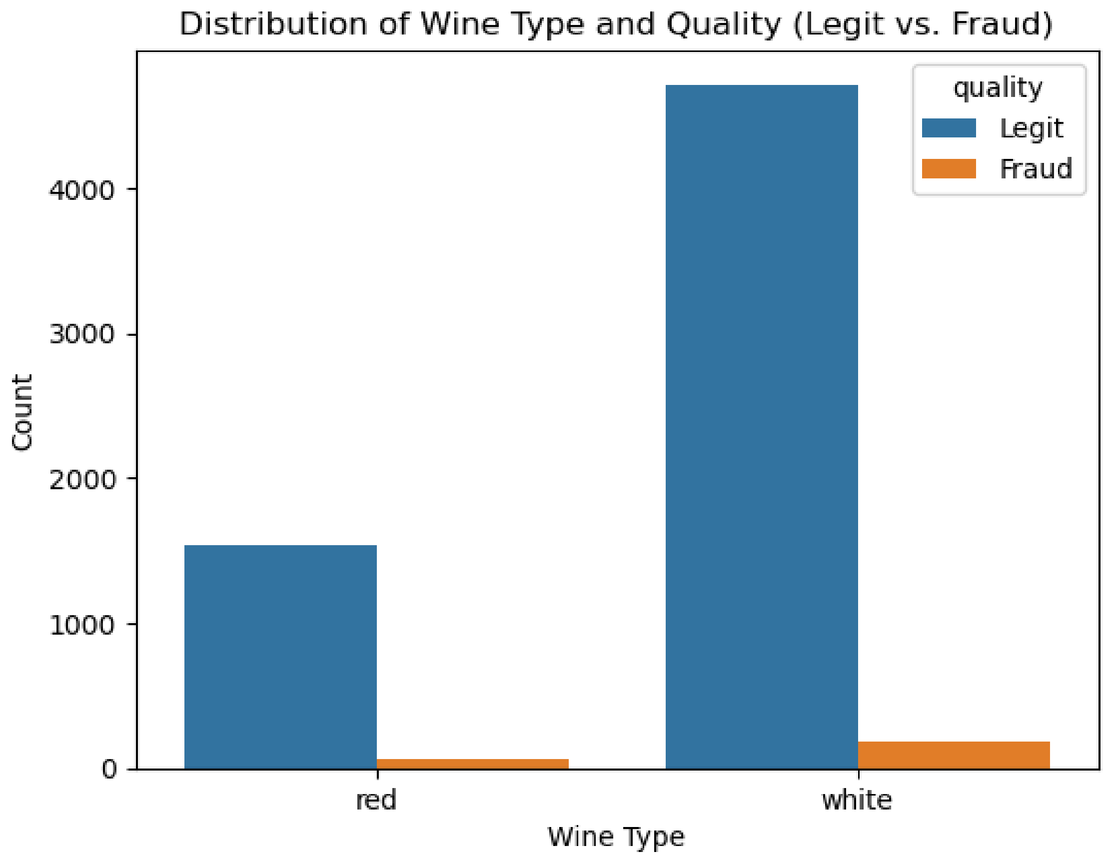


Task 4: Create a Countplot of Wine Type and Quality

Instruction: Plot a countplot to show the distribution of wine type (red or white) and classify them by "Legit" or "Fraud" using hue.

In [6]:
```
# Simple countplot for Wine Type and Quality using 'type' column
sns.countplot(x='type', hue='quality', data=df)
plt.title('Distribution of Wine Type and Quality (Legit vs. Fraud)')
plt.xlabel('Wine Type')
plt.ylabel('Count')
plt.show()

# This section completed by Lana Gilmore
```



Task 5: Calculate the Percentage of Fraud in Red and White Wines

Instruction: Calculate the percentage of wines labeled as "Fraud" for both red and white wines and print out the percentage for each wine type.

Hint: Filter the dataset by type and calculate the percentage of fraud cases for each subset.

In [7]:
```
# Calculate percentage of Fraud in Red wine
fraud_red = df[(df['type'] == 'red') & (df['quality'] == 'Fraud')]

total_red = df[df['type'] == 'red']

# Calculate the percentage of fraud cases
fraud_red_percentage = len(fraud_red) / len(total_red) * 100

print(f'Percentage of Fraud in Red Wines: {fraud_red_percentage:.2f}%')

# This section completed by Lana Gilmore
```

```
Percentage of Fraud in Red Wines: 3.94%
```

In [8]:
```
# Calculate percentage of Fraud in White wine
fraud_white = df[(df['type'] == 'white') & (df['quality'] == 'Fraud')]

total_white = df[df['type'] == 'white']

# Calculate the percentage of fraud cases
fraud_white_percentage = len(fraud_white) / len(total_white) * 100
print(f'Percentage of Fraud in White Wines: {fraud_white_percentage:.2f}%'

# This section completed by Lana Gilmore
```

```
Percentage of Fraud in White Wines: 3.74%
```

Task 6: Convert the Quality Column into a Numeric Format

Instruction: Convert the target variable `quality` from "Legit" and "Fraud" to 0 and 1, respectively, for the classification task.

In [9]:
```
# Convert 'Legit' to 0 and 'Fraud' to 1
df['quality'] = df['quality'].map({'Legit': 0, 'Fraud': 1})

# This section completed by Lana Gilmore
```

Task 7: Convert the Type Column into Dummy Variables

Instruction: Convert the categorical column `type` into numerical values using Pandas' `get_dummies()` function.

Hint: Use `drop_first=True` to avoid the dummy variable trap.

In [10]:
```
# Convert 'type' column into dummy variables with drop_first=True to avoid
df = pd.get_dummies(df, columns=['type'], drop_first=True)

# This section completed by Lana Gilmore
```

Task 8: Split the Dataset into Features and Target Variables

Instruction: Separate the features (X) from the target variable (y). Drop the `quality` column from X, and assign y to the Fraud column.

In [11]:
```
# Split features (X) and target variable (y)
X = df.drop(columns=['quality'])
y = df['quality']

# This section completed by Lana Gilmore
```

Task 9: Train-Test Split and Data Scaling

Instruction: Split the dataset into training and testing sets using an 80-20 split and random_state=42
Then, use `StandardScaler` to scale the features in both the training and testing sets.

Hint: Use `train_test_split` from Scikit-learn.

In [12]:
```
# Split the dataset into 80% training and 20% testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, r

# Scale the features using StandardScaler
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# This section completed by Lana Gilmore
```

Task 10: Train an SVM Model with Grid Search

Instruction: Use `GridSearchCV` to tune the SVM hyperparameters (C and gamma). Evaluate the best model using a confusion matrix and classification report.

- This code below uses GridSearchCV to find the best hyperparameters (C and gamma) for the SVM model and then evaluates the model using a confusion matrix and classification report.

In [13]:
```
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report, confusion_matrix

# Define parameter grid
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10], 'gamma': ['scale', 'auto']}
```

In [14]:
```
# Perform Grid Search with SVM
grid = GridSearchCV(SVC(class_weight='balanced'), param_grid, cv=5)
grid.fit(X_train, y_train)
```

Out[14]:
```
GridSearchCV (https://scikit-learn.org/1.5/modules/generated/sklearn.model_selection.GridSea
best_estimator_: SVC
SVC (https://scikit-learn.org/1.5/modules/generated/sklearn.svm.SVC.html)
```

In [15]:
```
# Get the best parameters
print("Best Parameters from GridSearchCV:")
print(grid.best_params_)
```

```
Best Parameters from GridSearchCV:
{'C': 10, 'gamma': 'auto'}
```

In [16]:
```
# Make predictions using the best model
grid_predictions = grid.predict(X_test)
```

Print the Confusion Matrix

In [17]:
```
conf_matrix = confusion_matrix(y_test, grid_predictions)
print("Confusion Matrix:")
print(conf_matrix)

# THis section completed by Lana Gilmore
```

```
Confusion Matrix:
[[1222   29]
 [  41    8]]
```

Print the Classification Report

In [18]:

```
class_report = classification_report(y_test, grid_predictions)
print("\nClassification Report:")
print(class_report)

# This section completed by Lana Gilmore
```

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

           0       0.97      0.98      0.97      1251
           1       0.22      0.16      0.19        49

    accuracy                           0.95      1300
   macro avg       0.59      0.57      0.58      1300
weighted avg       0.94      0.95      0.94      1300
```

Explain the classification report result. What do the numbers tell you?

The classification report shows that the model performs well in detecting legit transactions (Class 0) with high precision The report shows the model does a great job identifying legit transactions (97% precision and 98% recall). But when it comes to spotting fraud, it's not doing as well, with only 22% precision and 16% recall. So, while the overall accuracy is 95%, it's missing a lot of actual fraud cases and often predicting non-fraud as fraud. The fraud detection needs work.

Critical Thinking Question

Question: Based on the results of the classification report, how well did the model handle the imbalanced dataset?

Answer: The model struggled with handling the imbalanced dataset, which is clear from the classification report. It performed well with legit transactions, showing high precision and recall, meaning it correctly identified most legit cases. However, the performance on fraudulent transactions was poor, with low precision and recall. This shows the model missed many fraud cases and often predicted fraud when it wasn’t there. Since fraud cases are much rarer in this dataset, the model is biased toward predicting legit transactions correctly but doesn’t handle the minority class (fraud) effectively. This is a common issue with imbalanced data, and techniques like resampling or adjusting the model’s sensitivity to the minority class might help improve its performance.

Deep Learning Section: This section begins the implementation of deep learning models.

Neural Network Architecture

The neural network in this project is designed to analyze and predict wine quality based on its features. Below is an explanation of each component used in the architecture:

Input Layer:

This layer determines the number of input features based on the dataset. In this case, it corresponds to the various chemical properties of the wine used as predictors.

Hidden Layers:

These layers are responsible for learning complex patterns and relationships in the data by applying non-linear transformations. The first hidden layer has 64 neurons, while the second has 32 neurons. Both layers use the ReLU activation function to introduce non-linearity, allowing the model to handle more complex relationships.

Dropout Layers:

Dropout is a regularization technique that helps prevent overfitting by randomly "dropping" (disabling) a fraction of neurons during each training iteration. In this model, dropout layers follow each hidden layer, with a dropout rate of 30% (0.3), ensuring the model generalizes well to unseen data.

Output Layer:

The output layer is configured to classify wine quality into different categories. It uses the softmax activation function to output probabilities for each category, ensuring the predictions sum to 1. This is particularly useful for multi-class classification problems.

The neural network implementation is shown below, starting with data preprocessing and

In [19]:
```
# Importing required libraries and modules

# Sequential is used to define a linear stack of layers for the neural net
from tensorflow.keras.models import Sequential

# Dense: Fully connected layer; Dropout: Regularization layer; Input: Inpu
from tensorflow.keras.layers import Dense, Dropout, Input

# Adam optimizer for adaptive learning rate during training
from tensorflow.keras.optimizers import Adam

# Accuracy metric to evaluate the model's performance
from tensorflow.keras.metrics import Accuracy

# Utility function to convert labels to one-hot encoding for classificatio
from tensorflow.keras.utils import to_categorical

# LabelEncoder to encode target labels into numeric format for machine lea
from sklearn.preprocessing import LabelEncoder

# Function to split the dataset into training and testing sets
from sklearn.model_selection import train_test_split

# StandardScaler to standardize features by scaling them to zero mean and
from sklearn.preprocessing import StandardScaler

# Functions to evaluate the model's performance: classification report and
from sklearn.metrics import classification_report, confusion_matrix
```

In [20]:
```
# Load the dataset from a CSV file into a pandas DataFrame
df = pd.read_csv("data/wine_fraud.csv")
```

In [21]:
```
# Split the dataset into features (X) and target variable (y)
# Remove the 'quality' (target) and 'type' (irrelevant) columns from featu
X = df.drop(['quality', 'type'], axis=1)
y = df['quality']
```

In [22]:
```
# Encode the target variable ('quality') into numeric labels for classific
label_encoder = LabelEncoder()
y = label_encoder.fit_transform(y)
```

In [23]:
```
# Split the dataset into training and testing sets
# 20% of the data is reserved for testing, and random_state ensures reprod
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, r
```

In [24]:
```
# Standardize the features by scaling them to have zero mean and unit vari
# Fit the scaler to the training data and transform it
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)

# Apply the same transformation to the test data
X_test = scaler.transform(X_test)
```

In [25]:
```
# Define the neural network architecture using Sequential API
model = Sequential([
    Input(shape=(X_train.shape[1],)),  # Input layer with the same shape a
    Dense(64, activation='relu'),      # First hidden layer with 64 neuron
    Dropout(0.5),                      # Dropout layer to prevent overfitt
    Dense(32, activation='relu'),      # Second hidden layer with 32 neuro
    Dropout(0.3),                      # Dropout layer to prevent overfitt
    Dense(len(np.unique(y)), activation='softmax')  # Output layer with so
])
```

In [26]:
```
# Compile the model with the Adam optimizer and sparse categorical crossen
# Accuracy is used as the evaluation metric
model.compile(
    optimizer=Adam(learning_rate=0.001),  # Optimizer with a learning rate
    loss='sparse_categorical_crossentropy',  # Loss function for multi-cla
    metrics=['accuracy']  # Track accuracy during training
)
```

In [27]:
```
# Train the model on the training data
# Use 50 epochs, a batch size of 32, and reserve 20% of the data for valid
model.fit(
    X_train, y_train,
    epochs=50,
    batch_size=32,
    validation_split=0.2,
    verbose=2  # Display training progress for each epoch
)
```

```
Epoch 1/50
130/130 - 1s - 7ms/step - accuracy: 0.9456 - loss: 0.2494 - val_accuracy:
0.9615 - val_loss: 0.1643
Epoch 2/50
130/130 - 0s - 1ms/step - accuracy: 0.9620 - loss: 0.1776 - val_accuracy:
0.9615 - val_loss: 0.1579
Epoch 3/50
130/130 - 0s - 1ms/step - accuracy: 0.9615 - loss: 0.1829 - val_accuracy:
0.9615 - val_loss: 0.1514
Epoch 4/50
130/130 - 0s - 1ms/step - accuracy: 0.9622 - loss: 0.1678 - val_accuracy:
0.9615 - val_loss: 0.1499
Epoch 5/50
130/130 - 0s - 1ms/step - accuracy: 0.9622 - loss: 0.1695 - val_accuracy:
0.9615 - val_loss: 0.1480
Epoch 6/50
130/130 - 0s - 1ms/step - accuracy: 0.9622 - loss: 0.1628 - val_accuracy:
0.9615 - val_loss: 0.1447
Epoch 7/50
130/130 - 0s - 1ms/step - accuracy: 0.9622 - loss: 0.1602 - val_accuracy:
0.9615 - val_loss: 0.1422
Epoch 8/50
130/130 - 0s - 1ms/step - accuracy: 0.9625 - loss: 0.1511 - val_accuracy:
0.9615 - val_loss: 0.1418
Epoch 9/50
130/130 - 0s - 1ms/step - accuracy: 0.9620 - loss: 0.1553 - val_accuracy:
0.9615 - val_loss: 0.1394
Epoch 10/50
130/130 - 0s - 1ms/step - accuracy: 0.9625 - loss: 0.1588 - val_accuracy:
0.9615 - val_loss: 0.1392
Epoch 11/50
130/130 - 0s - 1ms/step - accuracy: 0.9622 - loss: 0.1520 - val_accuracy:
0.9615 - val_loss: 0.1403
Epoch 12/50
130/130 - 0s - 1ms/step - accuracy: 0.9625 - loss: 0.1487 - val_accuracy:
0.9615 - val_loss: 0.1378
Epoch 13/50
130/130 - 0s - 1ms/step - accuracy: 0.9625 - loss: 0.1512 - val_accuracy:
0.9615 - val_loss: 0.1386
Epoch 14/50
130/130 - 0s - 1ms/step - accuracy: 0.9620 - loss: 0.1476 - val_accuracy:
0.9615 - val_loss: 0.1379
Epoch 15/50
130/130 - 0s - 1ms/step - accuracy: 0.9620 - loss: 0.1464 - val_accuracy:
0.9615 - val_loss: 0.1364
Epoch 16/50
130/130 - 0s - 1ms/step - accuracy: 0.9620 - loss: 0.1474 - val_accuracy:
0.9615 - val_loss: 0.1354
Epoch 17/50
130/130 - 0s - 1ms/step - accuracy: 0.9622 - loss: 0.1472 - val_accuracy:
0.9606 - val_loss: 0.1362
Epoch 18/50
130/130 - 0s - 1ms/step - accuracy: 0.9620 - loss: 0.1463 - val_accuracy:
0.9615 - val_loss: 0.1355
Epoch 19/50
130/130 - 0s - 1ms/step - accuracy: 0.9620 - loss: 0.1458 - val_accuracy:
0.9615 - val_loss: 0.1339
```

```
Epoch 20/50
130/130 - 0s - 1ms/step - accuracy: 0.9618 - loss: 0.1429 - val_accuracy:
0.9615 - val_loss: 0.1312
Epoch 21/50
130/130 - 0s - 1ms/step - accuracy: 0.9627 - loss: 0.1418 - val_accuracy:
0.9615 - val_loss: 0.1304
Epoch 22/50
130/130 - 0s - 1ms/step - accuracy: 0.9615 - loss: 0.1401 - val_accuracy:
0.9615 - val_loss: 0.1314
Epoch 23/50
130/130 - 0s - 1ms/step - accuracy: 0.9627 - loss: 0.1405 - val_accuracy:
0.9615 - val_loss: 0.1317
Epoch 24/50
130/130 - 0s - 1ms/step - accuracy: 0.9627 - loss: 0.1422 - val_accuracy:
0.9615 - val_loss: 0.1296
Epoch 25/50
130/130 - 0s - 1ms/step - accuracy: 0.9625 - loss: 0.1386 - val_accuracy:
0.9615 - val_loss: 0.1311
Epoch 26/50
130/130 - 0s - 1ms/step - accuracy: 0.9620 - loss: 0.1394 - val_accuracy:
0.9615 - val_loss: 0.1297
Epoch 27/50
130/130 - 0s - 1ms/step - accuracy: 0.9630 - loss: 0.1385 - val_accuracy:
0.9615 - val_loss: 0.1277
Epoch 28/50
130/130 - 0s - 1ms/step - accuracy: 0.9639 - loss: 0.1325 - val_accuracy:
0.9615 - val_loss: 0.1264
Epoch 29/50
130/130 - 0s - 1ms/step - accuracy: 0.9639 - loss: 0.1356 - val_accuracy:
0.9606 - val_loss: 0.1273
Epoch 30/50
130/130 - 0s - 1ms/step - accuracy: 0.9630 - loss: 0.1373 - val_accuracy:
0.9615 - val_loss: 0.1272
Epoch 31/50
130/130 - 0s - 1ms/step - accuracy: 0.9630 - loss: 0.1330 - val_accuracy:
0.9606 - val_loss: 0.1267
Epoch 32/50
130/130 - 0s - 1ms/step - accuracy: 0.9618 - loss: 0.1407 - val_accuracy:
0.9606 - val_loss: 0.1274
Epoch 33/50
130/130 - 0s - 1ms/step - accuracy: 0.9630 - loss: 0.1346 - val_accuracy:
0.9606 - val_loss: 0.1265
Epoch 34/50
130/130 - 0s - 1ms/step - accuracy: 0.9622 - loss: 0.1330 - val_accuracy:
0.9606 - val_loss: 0.1280
Epoch 35/50
130/130 - 0s - 1ms/step - accuracy: 0.9627 - loss: 0.1343 - val_accuracy:
0.9596 - val_loss: 0.1268
Epoch 36/50
130/130 - 0s - 1ms/step - accuracy: 0.9639 - loss: 0.1317 - val_accuracy:
0.9615 - val_loss: 0.1278
Epoch 37/50
130/130 - 0s - 1ms/step - accuracy: 0.9620 - loss: 0.1369 - val_accuracy:
0.9615 - val_loss: 0.1276
Epoch 38/50
130/130 - 0s - 1ms/step - accuracy: 0.9618 - loss: 0.1375 - val_accuracy:
0.9606 - val_loss: 0.1261
```

```
Epoch 39/50
130/130 - 0s - 1ms/step - accuracy: 0.9620 - loss: 0.1305 - val_accuracy:
0.9606 - val_loss: 0.1245
Epoch 40/50
130/130 - 0s - 1ms/step - accuracy: 0.9630 - loss: 0.1369 - val_accuracy:
0.9615 - val_loss: 0.1258
Epoch 41/50
130/130 - 0s - 1ms/step - accuracy: 0.9625 - loss: 0.1358 - val_accuracy:
0.9606 - val_loss: 0.1253
Epoch 42/50
130/130 - 0s - 1ms/step - accuracy: 0.9630 - loss: 0.1317 - val_accuracy:
0.9615 - val_loss: 0.1259
Epoch 43/50
130/130 - 0s - 1ms/step - accuracy: 0.9622 - loss: 0.1321 - val_accuracy:
0.9615 - val_loss: 0.1254
Epoch 44/50
130/130 - 0s - 1ms/step - accuracy: 0.9622 - loss: 0.1314 - val_accuracy:
0.9606 - val_loss: 0.1229
Epoch 45/50
130/130 - 0s - 1ms/step - accuracy: 0.9627 - loss: 0.1312 - val_accuracy:
0.9606 - val_loss: 0.1231
Epoch 46/50
130/130 - 0s - 1ms/step - accuracy: 0.9632 - loss: 0.1321 - val_accuracy:
0.9615 - val_loss: 0.1237
Epoch 47/50
130/130 - 0s - 1ms/step - accuracy: 0.9620 - loss: 0.1310 - val_accuracy:
0.9606 - val_loss: 0.1230
Epoch 48/50
130/130 - 0s - 1ms/step - accuracy: 0.9634 - loss: 0.1298 - val_accuracy:
0.9606 - val_loss: 0.1240
Epoch 49/50
130/130 - 0s - 1ms/step - accuracy: 0.9622 - loss: 0.1287 - val_accuracy:
0.9606 - val_loss: 0.1246
Epoch 50/50
130/130 - 0s - 1ms/step - accuracy: 0.9618 - loss: 0.1335 - val_accuracy:
0.9615 - val_loss: 0.1256
```

Out[27]:
```
<keras.src.callbacks.history.History at 0x23bc9457fd0>
```

In [28]:
```
# Generate predictions on the test data
# np.argmax is used to extract the index of the class with the highest pro
y_pred = np.argmax(model.predict(X_test), axis=-1)

# Print the classification report to evaluate the model's performance
# Includes precision, recall, f1-score, and support for each class
print("Classification Report:")
print(classification_report(y_test, y_pred))
```

```
41/41 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
Classification Report:
              precision    recall  f1-score   support

           0       1.00      0.02      0.04        49
           1       0.96      1.00      0.98      1251

    accuracy                           0.96      1300
   macro avg       0.98      0.51      0.51      1300
weighted avg       0.96      0.96      0.95      1300
```

Classification Report Results

The classification report shows that the model performs well overall, achieving 96% accuracy, but struggles with class imbalance. For the majority class (`1`), the model demonstrates high precision (0.96) and recall (1.00), resulting in a strong F1-score of 0.98. However, it struggles significantly with the minority class (`0`), where precision is perfect at 1.00, but recall is extremely low at 0.02, meaning it correctly identifies only 2% of actual instances for this class.

This highlights how the model analyzes the chemical properties of wine to predict its quality category. While it is highly accurate for the majority class, it often misclassifies wines in the minority class as belonging to the majority. For wine quality prediction, this imbalance limits the model's reliability in identifying underrepresented quality categories, potentially leading to missed opportunities or inaccuracies for wines with unique characteristics. Addressing the imbalance through techniques like weighting classes or augmenting the dataset would be essential to improve performance across all quality categories.

Hyperparameter Tuning

This section experiments with different hyperparameters such as learning rate, batch size, and epochs to observe their impact on the model's performance. Hyperparameter tuning is essential to optimize the training process and achieve better results.

In [29]:
```
from tensorflow.keras.optimizers import Adam

# Experiment with learning rates
learning_rates = [0.001, 0.01, 0.0001]
batch_sizes = [16, 32, 64]
results = []

for lr in learning_rates:
    for batch in batch_sizes:
        print(f"Training model with learning rate={lr} and batch size={bat
        model.compile(optimizer=Adam(learning_rate=lr),
                      loss='sparse_categorical_crossentropy',
                      metrics=['accuracy'])
        history = model.fit(X_train, y_train,
                            epochs=5,  # Using fewer epochs for quick expe
                            batch_size=batch,
                            validation_split=0.2,
                            verbose=0)
        val_acc = history.history['val_accuracy'][-1]
        results.append((lr, batch, val_acc))

# Display the results of tuning
print("\nHyperparameter Tuning Results:")
for lr, batch, acc in results:
    print(f"Learning rate: {lr}, Batch size: {batch}, Validation accuracy:
```

```
Training model with learning rate=0.001 and batch size=16
Training model with learning rate=0.001 and batch size=32
Training model with learning rate=0.001 and batch size=64
Training model with learning rate=0.01 and batch size=16
Training model with learning rate=0.01 and batch size=32
Training model with learning rate=0.01 and batch size=64
Training model with learning rate=0.0001 and batch size=16
Training model with learning rate=0.0001 and batch size=32
Training model with learning rate=0.0001 and batch size=64

Hyperparameter Tuning Results:
Learning rate: 0.001, Batch size: 16, Validation accuracy: 0.9615
Learning rate: 0.001, Batch size: 32, Validation accuracy: 0.9615
Learning rate: 0.001, Batch size: 64, Validation accuracy: 0.9606
Learning rate: 0.01, Batch size: 16, Validation accuracy: 0.9615
Learning rate: 0.01, Batch size: 32, Validation accuracy: 0.9615
Learning rate: 0.01, Batch size: 64, Validation accuracy: 0.9615
Learning rate: 0.0001, Batch size: 16, Validation accuracy: 0.9615
Learning rate: 0.0001, Batch size: 32, Validation accuracy: 0.9615
Learning rate: 0.0001, Batch size: 64, Validation accuracy: 0.9615
```

Analysis of Impact

The hyperparameter tuning experiment shows that changes in learning rate and batch size have little effect on validation accuracy, which stays between 0.9606 and 0.9615. This suggests the model is stable across these settings and not highly sensitive to them.

However, the notebook doesn’t explore how these hyperparameters impact other metrics like training and validation loss or training time. Adding learning curves for different hyperparameter combinations could provide a clearer picture of their effects on the model's performance and

TensorBoard Integration

In [30]:

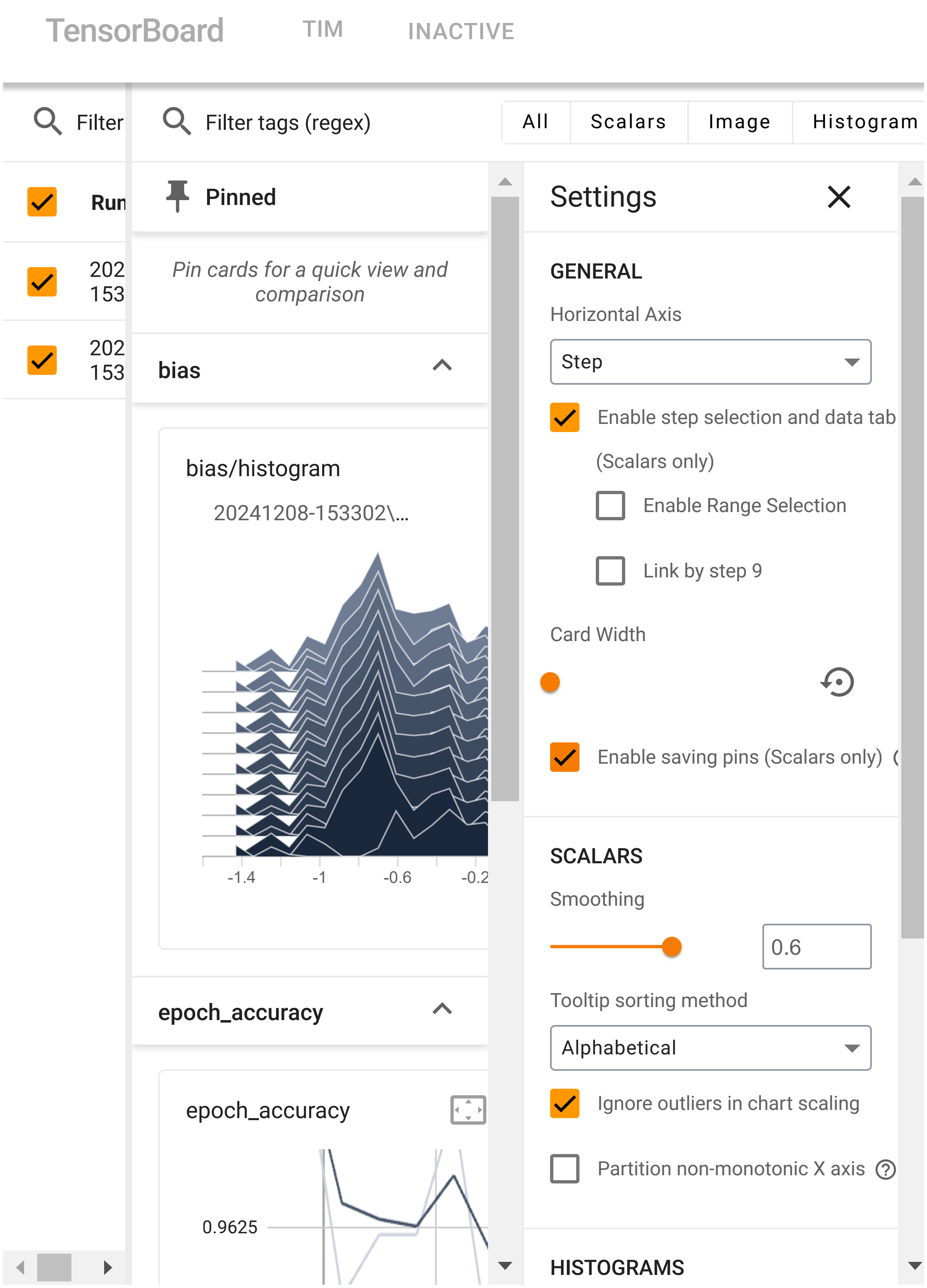
```
import tensorflow as tf
from tensorflow.keras.callbacks import TensorBoard
import datetime

# Set up a log directory for TensorBoard
log_dir = "logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback = TensorBoard(log_dir=log_dir, histogram_freq=1)

# Re-train the model with TensorBoard callback
model.fit(
    X_train, y_train,
    epochs=10,  # Adjust as needed
    batch_size=32,
    validation_split=0.2,
    callbacks=[tensorboard_callback]
)

# Launch TensorBoard in the notebook
%load_ext tensorboard
%tensorboard --logdir logs/fit
```

```
Epoch 1/10
130/130 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9624 - loss: 0.141
0 - val_accuracy: 0.9615 - val_loss: 0.1429
Epoch 2/10
130/130 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9649 - loss: 0.132
5 - val_accuracy: 0.9615 - val_loss: 0.1426
Epoch 3/10
130/130 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9658 - loss: 0.125
4 - val_accuracy: 0.9615 - val_loss: 0.1424
Epoch 4/10
130/130 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9642 - loss: 0.132
9 - val_accuracy: 0.9615 - val_loss: 0.1423
Epoch 5/10
130/130 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9580 - loss: 0.149
8 - val_accuracy: 0.9615 - val_loss: 0.1420
Epoch 6/10
130/130 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9621 - loss: 0.138
0 - val_accuracy: 0.9615 - val_loss: 0.1416
Epoch 7/10
130/130 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9619 - loss: 0.141
8 - val_accuracy: 0.9615 - val_loss: 0.1414
Epoch 8/10
130/130 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9605 - loss: 0.143
4 - val_accuracy: 0.9615 - val_loss: 0.1413
Epoch 9/10
130/130 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9615 - loss: 0.132
6 - val_accuracy: 0.9615 - val_loss: 0.1411
Epoch 10/10
130/130 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9632 - loss: 0.134
5 - val_accuracy: 0.9615 - val_loss: 0.1411
```


TensorBoard
TIM
INACTIVE
Filter
Filter tags (regex)
All
Scalars
Image
Histogram
Run
202
153
202
153
Pinned
Pin cards for a quick view and comparison
bias
bias/histogram
20241208-153302\...
-1.4
-1
-0.6
-0.2
epoch_accuracy
epoch_accuracy
0.9625
Settings
GENERAL
Horizontal Axis
Step
Enable step selection and data tab
(Scalars only)
Enable Range Selection
Link by step 9
Card Width
Enable saving pins (Scalars only)
SCALARS
Smoothing
0.6
Tooltip sorting method
Alphabetical
Ignore outliers in chart scaling
Partition non-monotonic X axis
HISTOGRAMS

Conclusions

Updated Conclusions

This project focused on predicting wine quality using machine learning and deep learning techniques, leveraging chemical properties as features. The results and additional tools, such as hyperparameter tuning and TensorBoard integration, provided valuable insights into model performance.

The **Support Vector Machine (SVM)** achieved 95% accuracy, performing well for the majority class (`0`) with precision (0.97) and recall (0.98). However, it struggled with the minority class (`1`), showing low recall (0.16) and an F1-score of 0.19. This highlights the impact of class imbalance on the SVM's ability to generalize.

The **Neural Network** demonstrated slightly higher accuracy at 96%, excelling for the majority class (`1`) with perfect recall (1.00) and an F1-score of 0.98. However, its performance on the minority class (`0`) was weaker, with a recall of 0.02 and an F1-score of 0.04, showing significant bias toward the dominant class.

Hyperparameter tuning experiments tested various learning rates and batch sizes. The results showed minimal impact on validation accuracy, which consistently ranged from 0.9606 to 0.9615, indicating model stability across tested combinations. TensorBoard integration provided additional visualization tools to track training and validation performance, offering valuable insights into model optimization and generalization trends.

Key Takeaways:

- The chemical properties of wine proved effective for predicting quality, especially for the majority class.
- Class imbalance negatively affected both models, emphasizing the need for oversampling, class weighting, or synthetic data to improve minority class predictions.
- Hyperparameter tuning showed stability in accuracy, but deeper analysis of its impact on training and validation loss is needed.
- TensorBoard served as a useful tool for monitoring training metrics and improving understanding of the optimization process.

In conclusion, both models effectively utilized the chemical features to predict wine quality, but addressing class imbalance remains crucial for improving overall performance and ensuring accurate predictions across all quality categories.

Credits

- The machine learning section has been updated for this project using content from a previous "SVM Lab Assignment." The original link for the assignment is located on a local host server: SVM Lab Assignment (http://localhost:8888/notebooks/Desktop/CISB%2060%20ANGEL/MODULE%205/SVM%20

A copy of the original file can be found in the data subfolder for reference.

- Deep learning section adapted from the "Neural Network" code provided by Professor Angel Hernandez of Mt. San Antonio College for the course CISB 60 – Machine Learning and Deep Learning (Fall 2024). YouTube link for reference: https://youtu.be/8WlXzOHN_Bo?si=EsuGBQOeGtV9yuC (https://youtu.be/8WlXzOHN_Bo?si=EsuGBQOeGtV9yuC)

- All other codes used in this project may have been adapted from academic topics covered throughout CISB 60 – Machine Learning and Deep Learning (Fall 2024).