A2.10 Neural Networks

Neural Networks and Feature Scaling

Neural networks, an integral part of modern machine learning techniques, derive several benefits from feature scaling. Feature scaling standardises the range of input data, ensuring that all the features have a similar scale. There are several key reasons why this is crucial for the efficient training of neural networks:

1. Convergence: Neural networks frequently employ optimisation techniques, such as gradient descent, to adjust weights while training. For these algorithms to achieve their goal faster and more consistently, it's vital for the input data to be normalised. If left unscaled, the optimisation algorithms might take longer or not converge.

2. Weight Updates: When all features are scaled appropriately, they contribute evenly to the update of the weights during the learning process. If not, features with larger numerical values could overshadow the rest, making the network overly reliant on them and potentially neglecting other essential features.

3. Numerical Stability: Deep neural networks can sometimes face issues related to numerical instability. Large input values can result in large intermediate calculations within the network, leading to challenges like vanishing or exploding gradients. Through scaling, these complications can be substantially reduced.

4. Regularisation: Regularisation techniques such as L1 and L2 are susceptible to the magnitude of input features. Proper scaling ensures that the regularisation impact is distributed uniformly across all features.L1 regularisation, often referred to as Lasso regression, has been known to encourage sparsity in the model by adding a penalty equivalent to the absolute value of the magnitude of coefficients. On the other hand, L2 regularisation, commonly termed Ridge regression, adds a penalty equivalent to the square of the magnitude of coefficients. Both techniques are sensitive to the scale of input features. Through scaling, it is ensured that these techniques are applied uniformly across features.

Notably, in many conventional machine learning methods, introducing feature scaling doesn't typically give rise to concerns surrounding data leakage. Data leakage is when the test dataset's information unintentionally infiltrates the training process. Such inadvertent blending can skew the model's performance metrics, leading to a misleadingly positive evaluation of its capability. In contrast, in traditional models, the scaling process has no bearing on the decision boundaries or rules. This independence allows for the entire dataset to be scaled without reservation. While traditional machine learning models remain impervious to the effects of scaling on their decision-making processes, neural networks present a more complex interplay. Given their iterative learning and weight adjustment processes, even subtle leaks of test data information can profoundly distort the learning dynamics. This underscores the necessity for meticulous dataset handling and scaling methodologies tailored explicitly for neural networks to safeguard against such pitfalls.

Neural networks adjust their weights in the training phase; there is a potential risk: scaling both training and test data collectively could inadvertently allow information from the test dataset to influence the training phase. Such a scenario might culminate in overfitting. To circumvent the pitfalls of data leakage, it has been proposed that training data be scaled independently and the identical scaling parameters be subsequently applied to the test data. The demarcation between the training and test datasets is preserved through such a procedure, thus facilitating a more accurate and unbiased model evaluation.

```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

The provided code demonstrates a common preprocessing step when working with data in machine learning: feature scaling using standardisation. Let's break down what each line is doing:

1. **`scaler = StandardScaler()`**:
   - This line initialises an instance of the `StandardScaler` class from the `scikit-learn` library. The `StandardScaler` is used to standardise features by removing the mean and scaling them to unit variance. This process is also called z-score normalisation. The formula for this is:
     \[z = \frac{x - \mu}{\sigma} \]
     where \(x \) is the original feature vector, \(\mu \) is the mean of the feature, and \(\sigma \) is its standard deviation.

2. **`X_train_scaled = scaler.fit_transform(X_train)`**:
   - The `fit_transform` method does two things:
     1. **`fit`**: It computes the mean (\(\mu \)) and standard deviation (\(\sigma \)) of the data in `X_train`. These values are then stored internally in the `scaler` object.
     2. **`transform`**: It scales the `X_train` data using the previously computed mean and standard deviation. This transformed data is stored in `X_train_scaled`.

3. **`X_test_scaled = scaler.transform(X_test)`**:
   - Here, the `transform` method scales the `X_test` data using the mean and standard deviation computed from the `X_train` data. It's crucial to use the statistics from the training set to scale the test set to ensure consistency in the preprocessing steps between training and testing phases. This helps prevent data leakage, where information from the test set might inadvertently be used during training.

Feature scaling, especially standardisation, is essential for many machine learning algorithms. Algorithms like Support Vector Machines, k-Nearest Neighbours, and many neural networks rely on distances between data points. If features have different scales, the distance-based algorithms may not perform as expected. By scaling the features, we ensure that each feature contributes equally to the computation of distances or optimisation processes.

```
fig, ax = plt.subplots(ncols=2, figsize=(15,5))
sns.distplot(np.concatenate(X_train), ax=ax[0]).set_title('Original Data')
sns.distplot(np.concatenate(X_train_scaled), ax=ax[1]).set_title('Scaled Data')
plt.tight_layout
plt.show()
```

The code provided aims to visualise the distributions of original data and scaled data side by side using Python's `matplotlib` and `seaborn` libraries. Here's an explanation of what each line does:

1. **`fig, ax = plt.subplots(ncols=2, figsize=(15,5))`**:
   - This line initializes a new `matplotlib` figure with two subplots arranged in one row and two columns, making it easy to compare two plots side by side. The figure size is set to be 15 units wide and 5 units tall. The `fig` variable contains the entire figure, and `ax` is an array containing the axes for the two subplots.

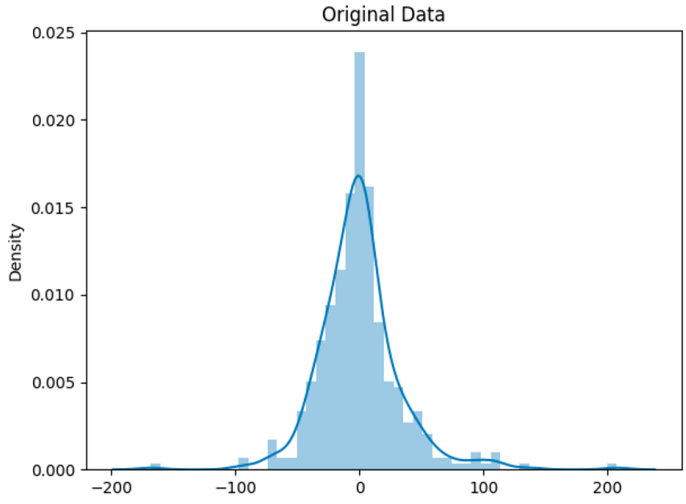
2. **`sns.distplot(np.concatenate(X_train), ax=ax[0]).set_title('Original Data')`**:
   - This line plots the distribution of the `X_train` data on the first subplot.
   - `np.concatenate(X_train)` is used to flatten the `X_train` data if it's a multi-dimensional array.
   - The `sns.distplot` function from the `seaborn` library plots a histogram with a line that estimates the probability density function of the data.
   - `.set_title('Original Data')` sets the title of the first subplot to "Original Data".

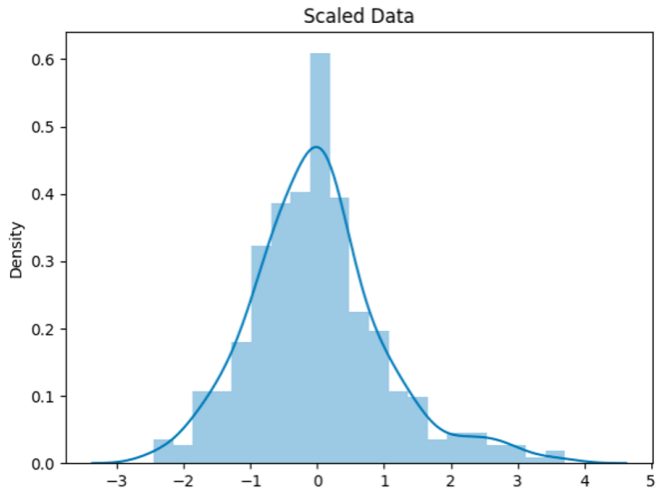
3. **`sns.distplot(np.concatenate(X_train_scaled), ax=ax[1]).set_title('Scaled Data')`**:
   - This line plots the distribution of the `X_train_scaled` data on the second subplot.
   - As before, `np.concatenate` is used to flatten the data if necessary.
   - The title of the second subplot is set to "Scaled Data".

4. **`plt.tight_layout`**:
   - This function adjusts the spacing between subplots to reduce overlaps and fit the subplots within the figure area. However, there's a minor mistake in the code: it should be `plt.tight_layout()` (with parentheses) to actually call the function.

5. **`plt.show()`**:
   - This line displays the figure. If you're working within an interactive environment like a Jupyter Notebook, this will render the plot inline. In some other environments or scripts, this command is essential to display the plot.

In essence, after executing this code, one should see two side-by-side plots: one displaying the distribution of the original data and the other showing the distribution of the standardized (scaled) data. This visualization helps in understanding the effect of the standard scaling on the data distribution.


Original Data
Density
0.025
0.020
0.015
0.010
0.005
0.000
−200
−100
0
100
200

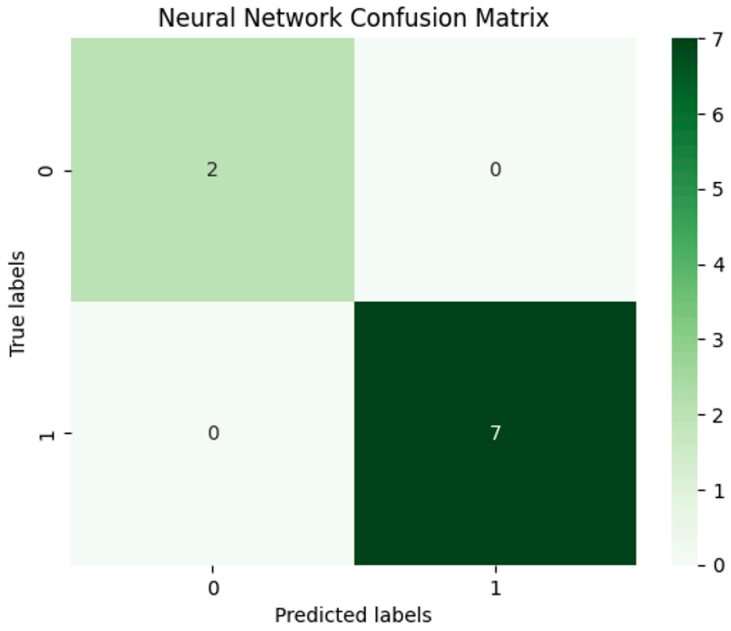

Scaled Data
Density
0.6
0.5
0.4
0.3
0.2
0.1
0.0
−3
−2
−1
0
1
2
3
4
5

```
[ ] NN_model = keras.Sequential([
        layers.Dense(16, activation='relu', input_shape=X_train_scaled[1].shape),
        layers.Dense(8, activation='relu'),
        layers.Dense(1, activation='sigmoid')
    ])

    NN_model.compile(
        loss='binary_crossentropy',
        optimizer='adam',
        metrics=['binary_accuracy']
    )

    early_stopping = keras.callbacks.EarlyStopping(
        patience=10,
        min_delta=0.0005,
        restore_best_weights=True,
    )


    train_history = NN_model.fit(
        X_train_scaled, y_train,
        validation_data=(X_test_pca, y_test),
        batch_size = 4,
        epochs = 200,
        callbacks=[early_stopping]
    )
```

```
Epoch 1/200
5/5 [==============================] - 3s 138ms/step - loss: 0.6760 - binary_accuracy: 0.6500 - val_loss: 4.0554 - val_bi
Epoch 2/200
5/5 [==============================] - 0s 18ms/step - loss: 0.6476 - binary_accuracy: 0.6500 - val_loss: 3.8747 - val_bir
Epoch 3/200
5/5 [==============================] - 0s 19ms/step - loss: 0.6258 - binary_accuracy: 0.7000 - val_loss: 3.6704 - val_bir
Epoch 4/200
5/5 [==============================] - 0s 21ms/step - loss: 0.6067 - binary_accuracy: 0.7500 - val_loss: 3.4790 - val_bir
Epoch 5/200
5/5 [==============================] - 0s 23ms/step - loss: 0.5880 - binary_accuracy: 0.8000 - val_loss: 3.2719 - val_bir
Epoch 6/200
5/5 [==============================] - 0s 27ms/step - loss: 0.5728 - binary_accuracy: 0.8000 - val_loss: 3.0759 - val_bir
Epoch 7/200
```

```
5/5 [==============================] - 0s 46ms/step - loss: 0.5566 - binary_accuracy: 0.8000 - val_loss: 2.8737 - val_bir
Epoch 8/200
5/5 [==============================] - 0s 29ms/step - loss: 0.5424 - binary_accuracy: 0.8500 - val_loss: 2.6974 - val_bir
Epoch 9/200
5/5 [==============================] - 0s 26ms/step - loss: 0.5275 - binary_accuracy: 0.8500 - val_loss: 2.5720 - val_bir
Epoch 10/200
5/5 [==============================] - 0s 20ms/step - loss: 0.5149 - binary_accuracy: 0.8500 - val_loss: 2.4457 - val_bir
Epoch 11/200
5/5 [==============================] - 0s 26ms/step - loss: 0.5019 - binary_accuracy: 0.8500 - val_loss: 2.3316 - val_bir
Epoch 12/200
5/5 [==============================] - 0s 15ms/step - loss: 0.4884 - binary_accuracy: 0.8500 - val_loss: 2.2518 - val_bir
Epoch 13/200
5/5 [==============================] - 0s 15ms/step - loss: 0.4758 - binary_accuracy: 0.8500 - val_loss: 2.1677 - val_bir
Epoch 14/200
5/5 [==============================] - 0s 21ms/step - loss: 0.4643 - binary_accuracy: 0.8500 - val_loss: 2.0839 - val_bir
Epoch 15/200
5/5 [==============================] - 0s 19ms/step - loss: 0.4521 - binary_accuracy: 0.8500 - val_loss: 2.0096 - val_bir
Epoch 16/200
5/5 [==============================] - 0s 29ms/step - loss: 0.4403 - binary_accuracy: 0.8500 - val_loss: 1.9364 - val_bir
Epoch 17/200
5/5 [==============================] - 0s 51ms/step - loss: 0.4285 - binary_accuracy: 0.8500 - val_loss: 1.8642 - val_bir
Epoch 18/200
5/5 [==============================] - 0s 30ms/step - loss: 0.4171 - binary_accuracy: 0.8500 - val_loss: 1.7981 - val_bir
Epoch 19/200
5/5 [==============================] - 0s 30ms/step - loss: 0.4065 - binary_accuracy: 0.8500 - val_loss: 1.7408 - val_bir
Epoch 20/200
5/5 [==============================] - 0s 29ms/step - loss: 0.3950 - binary_accuracy: 0.8500 - val_loss: 1.6739 - val_bir
Epoch 21/200
5/5 [==============================] - 0s 31ms/step - loss: 0.3850 - binary_accuracy: 0.8500 - val_loss: 1.6090 - val_bir
Epoch 22/200
5/5 [==============================] - 0s 32ms/step - loss: 0.3748 - binary_accuracy: 0.8500 - val_loss: 1.5411 - val_bir
Epoch 23/200
5/5 [==============================] - 0s 42ms/step - loss: 0.3651 - binary_accuracy: 0.8500 - val_loss: 1.4783 - val_bir
Epoch 24/200
5/5 [==============================] - 0s 31ms/step - loss: 0.3552 - binary_accuracy: 0.8500 - val_loss: 1.3924 - val_bir
Epoch 25/200
5/5 [==============================] - 0s 39ms/step - loss: 0.3455 - binary_accuracy: 0.8500 - val_loss: 1.3122 - val_bir
Epoch 26/200
5/5 [==============================] - 0s 34ms/step - loss: 0.3374 - binary_accuracy: 0.9000 - val_loss: 1.2484 - val_bir
Epoch 27/200
5/5 [==============================] - 0s 29ms/step - loss: 0.3288 - binary_accuracy: 0.9000 - val_loss: 1.2317 - val_bir
Epoch 28/200
5/5 [==============================] - 0s 28ms/step - loss: 0.3208 - binary_accuracy: 0.9000 - val_loss: 1.2180 - val_bir
Epoch 29/200
```

```
pred = NN_model.predict(X_test_scaled)
y_pred = np.round(pred)

print('Neural Network accuracy: ', round(accuracy_score(y_test, y_pred), 3))

# Confusion Matrix
cm_nn = confusion_matrix(y_test, y_pred)
ax = plt.subplot()
sns.heatmap(cm_nn, annot=True, ax=ax, fmt='g', cmap='Greens')
ax.set_xlabel('Predicted labels')
ax.set_ylabel('True labels')
ax.set_title('Neural Network Confusion Matrix')
plt.show()

# Precision, Recall, and F1 Score for class 0
precision_class_0 = precision_score(y_test, y_pred, pos_label=0)
recall_class_0 = recall_score(y_test, y_pred, pos_label=0)
f1_class_0 = f1_score(y_test, y_pred, pos_label=0)

# Precision, Recall, and F1 Score for class 1
precision_class_1 = precision_score(y_test, y_pred, pos_label=1)
recall_class_1 = recall_score(y_test, y_pred, pos_label=1)
f1_class_1 = f1_score(y_test, y_pred, pos_label=1)

print(f'Class 0 Precision: {precision_class_0:.4f}, Recall:
{recall_class_0:.4f}, F1 Score: {f1_class_0:.4f}')
print(f'Class 1 Precision: {precision_class_1:.4f}, Recall:
{recall_class_1:.4f}, F1 Score: {f1_class_1:.4f}')
```




```
Class 0 Precision: 1.0000, Recall: 1.0000, F1 Score: 1.0000
Class 1 Precision: 1.0000, Recall: 1.0000, F1 Score: 1.0000
```

This code visualises a confusion matrix for the predictions made by the neural network model on the test dataset using the Seaborn library. This code visualises the confusion matrix for the neural network model's predictions on the test set in a heatmap format. The heatmap provides an intuitive representation of the model's performance regarding correctly and incorrectly classified instances.

1. **Confusion Matrix Calculation**:

   `cm_nn = confusion_matrix(y_test, y_pred)`

   This line calculates the confusion matrix using the actual test labels (`y_test`) and the predicted labels (`y_pred`). The resulting `cm_nn` matrix will count true positives, true negatives, false positives, and false negatives.

2. **Plot Initialisation**:

   `ax = plt.subplot()`

   Here, a new subplot is initialized and its axis object is stored in the variable `ax`.

3. **Drawing the Heatmap**:

   `sns.heatmap(cm_nn, annot=True, ax = ax, fmt='g', cmap='Greens')`

   The Seaborn library's `heatmap` function is used to visualize the confusion matrix. Here's a breakdown of the parameters:

   - `cm_nn`: The confusion matrix data.
   - `annot=True`: Annotate each cell with the numeric value.
   - `ax = ax`: Use the previously created axis to plot the heatmap.
   - `fmt='g'`: The format for string conversion. In this context, 'g' means to format the number in the most compact representation.
   - `cmap='Greens'`: Use the green colour map to colour the cells. The darker the shade, the higher the number in the cell.

4. **Labels, Title, and Ticks**:

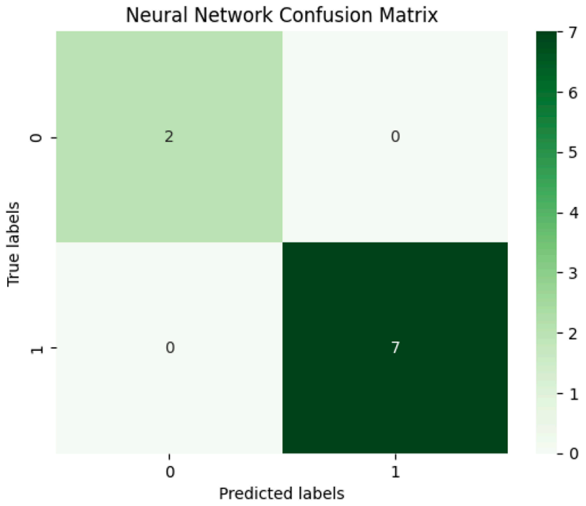
   The following lines label the axes and title the heatmap:

   - `ax.set_xlabel('Predicted labels')`: Labels the x-axis.
   - `ax.set_ylabel('True labels')`: Labels the y-axis.
   - `ax.set_title('Neural Network Confusion Matrix')`: Sets the title of the heatmap.

```
    from keras import layers, models, callbacks
    import tensorflow as tf
    from sklearn.metrics import accuracy_score, confusion_matrix
    import seaborn as sns
    import matplotlib.pyplot as plt

    # Define the class weights
    class_weight = {0: len(y_train) / (2 * sum(y_train == 0)), 1: len(y_train) / (2 * sum(y_train == 1))}

    # Build the Neural Network model
    NN_model = models.Sequential([
        layers.Dense(16, activation='relu', input_shape=X_train_scaled[1].shape),
        layers.Dense(8, activation='relu'),
        layers.Dense(1, activation='sigmoid')
    ])

    NN_model.compile(
        loss='binary_crossentropy',
        optimizer='adam',
        metrics=['binary_accuracy']
    )

    early_stopping = callbacks.EarlyStopping(
        patience=10,
        min_delta=0.0005,
        restore_best_weights=True,
    )

    # Train the model with class weight balancing
    train_history = NN_model.fit(
        X_train_scaled, y_train,
        validation_data=(X_test_pca, y_test),
        batch_size=4,
        epochs=200,
        callbacks=[early_stopping],
        class_weight=class_weight
    )


# Make predictions on the test set
pred = NN_model.predict(X_test_pca)
y_pred = [round(i[0]) for i in pred]

# Calculate and print accuracy
accuracy = accuracy_score(y_test, y_pred)
print('Neural Network accuracy:', round(accuracy, 3))

# Generate and display confusion matrix
cm_nn = confusion_matrix(y_test, y_pred)
ax = plt.subplot()
sns.heatmap(cm_nn, annot=True, ax=ax, fmt='g', cmap='Greens')

# Labels, title, and ticks
ax.set_xlabel('Predicted labels')
ax.set_ylabel('True labels')
ax.set_title('Neural Network Confusion Matrix')

plt.show()
```

```
# Precision, Recall, and F1 Score for class 0
precision_class_0 = precision_score(y_test, y_pred, pos_label=0)
recall_class_0 = recall_score(y_test, y_pred, pos_label=0)
f1_class_0 = f1_score(y_test, y_pred, pos_label=0)

# Precision, Recall, and F1 Score for class 1
precision_class_1 = precision_score(y_test, y_pred, pos_label=1)
recall_class_1 = recall_score(y_test, y_pred, pos_label=1)
f1_class_1 = f1_score(y_test, y_pred, pos_label=1)
```

```
print(f'Class 0 Precision: {precision_class_0:.4f}, Recall:
{recall_class_0:.4f}, F1 Score: {f1_class_0:.4f}')
print(f'Class 1 Precision: {precision_class_1:.4f}, Recall:
{recall_class_1:.4f}, F1 Score: {f1_class_1:.4f}')
```




```
Class 0 Precision: 1.0000, Recall: 1.0000, F1 Score: 1.0000
Class 1 Precision: 1.0000, Recall: 1.0000, F1 Score: 1.0000
```

This code is about building, training, and evaluating a neural network using the Keras library with TensorFlow as the backend, and then visualising its performance using a confusion matrix. Here's a step-by-step breakdown:

1. Import Necessary Libraries:

Relevant modules from Keras, TensorFlow, and Scikit-learn are imported. Additionally, Seaborn and Matplotlib are imported for visualisation.

2. Define Class Weights:

`class_weight = {0: len(y_train) / (2 * sum(y_train == 0)), 1: len(y_train) / (2 * sum(y_train == 1))}`

This code calculates the class weights to tackle any class imbalance. The idea is to give more weight to the underrepresented class so that during training, the neural network pays more attention to it. The weights are calculated based on the proportion of each class in the training data.

3. Build the Neural Network Model:

The `Sequential` model is constructed with three layers: two hidden layers followed by an output layer. Activation functions for the hidden layers are ReLU, and for the output layer, it's sigmoid because it's a binary classification problem.

4. Model Compilation:

The model is compiled with binary cross-entropy as the loss function, Adam optimiser, and binary accuracy as the metric.

5. Set Early Stopping Callback:

An early stopping callback is defined to halt training if the model doesn't improve beyond a certain threshold (`min_delta`) for a certain number of epochs (`patience`). This helps in preventing overfitting and saves computational resources.

6. Train the Model:

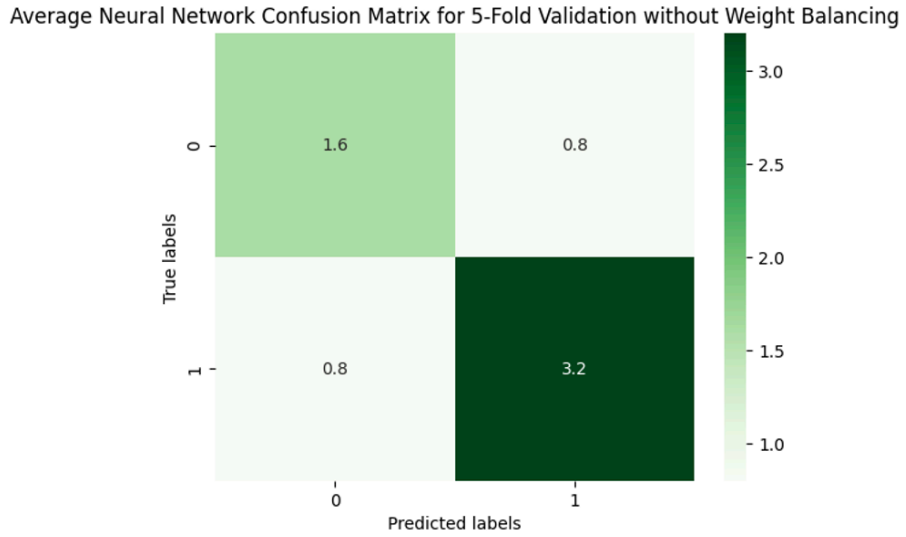
The neural network model is trained using the PCA-transformed training data. Class weights are used to handle any class imbalances. The model's performance is also validated against a separate validation dataset.

7. Make Predictions:

Once the model is trained, it's used to make predictions on the test set. These predictions, which are in the form of probabilities, are then rounded off to get binary class labels (0 or 1).

```
# K-Fold Validation without Class balancing
from keras import layers, models, callbacks
import tensorflow as tf
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import accuracy_score, confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

# Build the Neural Network model
NN_model = models.Sequential([
    layers.Dense(16, activation='relu', input_shape=X_train_scaled[1].shape),
    layers.Dense(8, activation='relu'),
    layers.Dense(1, activation='sigmoid')
])

NN_model.compile(
    loss='binary_crossentropy',
    optimizer='adam',
    metrics=['binary_accuracy']
)

# Define the number of folds (k) for k-Fold validation
k = 5

# Initialize the StratifiedKFold object
skf = StratifiedKFold(n_splits=k, shuffle=True, random_state=42)

# Lists to store accuracy and confusion matrices for each fold
accuracies = []
confusion_matrices = []

# Perform k-Fold validation
for train_index, test_index in skf.split(X_train_scaled, y_train):
    X_train_fold, X_test_fold = X_train_scaled[train_index], X_train_scaled[test_index]
    y_train_fold, y_test_fold = y_train[train_index], y_train[test_index]

    early_stopping = callbacks.EarlyStopping(
        patience=10,
        min_delta=0.0005,
```

```
        restore_best_weights=True,
    )

    # Train the model without class weight balancing
    train_history = NN_model.fit(
        X_train_fold, y_train_fold,
        validation_data=(X_test_fold, y_test_fold),
        batch_size=4,
        epochs=200,
        callbacks=[early_stopping]
    )

    # Make predictions on the test set
    pred = NN_model.predict(X_test_fold)
    y_pred = [round(i[0]) for i in pred]

    # Calculate and store accuracy
    accuracy = accuracy_score(y_test_fold, y_pred)
    accuracies.append(accuracy)

    # Generate confusion matrix for this fold and store it
    cm_nn = confusion_matrix(y_test_fold, y_pred)
    confusion_matrices.append(cm_nn)

# Calculate the average accuracy across all folds
average_accuracy = sum(accuracies) / len(accuracies)
print('Average Neural Network accuracy with 5-Fold Validation:', round(average_accuracy, 3))

# Calculate the average confusion matrix across all folds
average_cm = sum(confusion_matrices) / len(confusion_matrices)
ax = plt.subplot()
sns.heatmap(average_cm, annot=True, ax=ax, fmt='g', cmap='Greens')

# Labels, title, and ticks
ax.set_xlabel('Predicted labels')
ax.set_ylabel('True labels')
ax.set_title('Average Neural Network Confusion Matrix for 5-Fold Validation without Weight Balancing')

plt.show()



# Precision, Recall, and F1 Score for class 0
precision_class_0 = precision_score(y_test_fold, y_pred, pos_label=0)
recall_class_0 = recall_score(y_test_fold, y_pred, pos_label=0)
f1_class_0 = f1_score(y_test_fold, y_pred, pos_label=0)

# Precision, Recall, and F1 Score for class 1
precision_class_1 = precision_score(y_test_fold, y_pred, pos_label=1)
recall_class_1 = recall_score(y_test_fold, y_pred, pos_label=1)
f1_class_1 = f1_score(y_test_fold, y_pred, pos_label=1)

print(f'Class 0 Precision: {precision_class_0:.4f}, Recall:
{recall_class_0:.4f}, F1 Score: {f1_class_0:.4f}')
print(f'Class 1 Precision: {precision_class_1:.4f}, Recall:
{recall_class_1:.4f}, F1 Score: {f1_class_1:.4f}')
```




```
Class 0 Precision: 1.0000, Recall: 1.0000, F1 Score: 1.0000
Class 1 Precision: 1.0000, Recall: 1.0000, F1 Score: 1.0000
```

This code block conducts a 5-Fold Cross-Validation without class balancing on a dataset using a Neural Network, as detailed below:

1. Import Necessary Libraries and Modules:
```python
from keras import layers, models, callbacks
import tensorflow as tf
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import accuracy_score, confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt
```
2. Building the Neural Network:
```python
NN_model = models.Sequential([
    layers.Dense(16, activation='relu', input_shape=X_train_pca[1].shape),
    layers.Dense(8, activation='relu'),
    layers.Dense(1, activation='sigmoid')
])
```
- A simple neural network with 3 layers is built: two hidden layers and one output layer.
- `input_shape=X_train_pca[1].shape` indicates the shape of the input data.

```python
NN_model.compile(
    loss='binary_crossentropy',
    optimizer='adam',
    metrics=['binary_accuracy']
)
```
- The model is compiled using a binary crossentropy loss (ideal for binary classification tasks), the Adam optimiser, and will track binary accuracy as a metric.

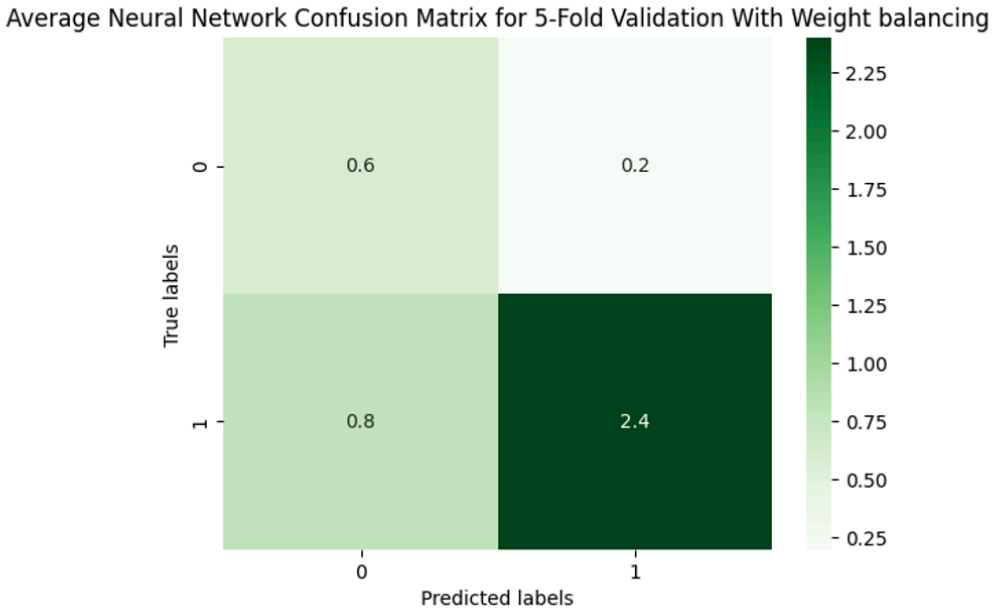
3. Set Up for k-Fold Cross-Validation:
```python
k = 5
skf = StratifiedKFold(n_splits=k, shuffle=True, random_state=42)
```
- 5-fold cross-validation is set up using `StratifiedKFold`, which maintains the class distribution across each fold.
- Data is shuffled before splitting, and a `random_state` is set for reproducibility.

4. Lists for Storing Results:
```python
accuracies = []
confusion_matrices = []
```

```
from keras import layers, models, callbacks
import tensorflow as tf
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import accuracy_score, confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

# Define the class weights
class_weight = {0: len(y_train) / (2 * sum(y_train == 0)), 1: len(y_train) / (2 * sum(y_train == 1))}

# Build the Neural Network model
NN_model = models.Sequential([
    layers.Dense(16, activation='relu', input_shape=X_train_scaled[1].shape),
    layers.Dense(8, activation='relu'),
    layers.Dense(1, activation='sigmoid')
])

NN_model.compile(
    loss='binary_crossentropy',
    optimizer='adam',
    metrics=['binary_accuracy']
)

# Define the number of folds (k) for k-Fold validation
k = 5

# Initialize the StratifiedKFold object
skf = StratifiedKFold(n_splits=k, shuffle=True, random_state=42)

# Lists to store accuracy and confusion matrices for each fold
accuracies = []
confusion_matrices = []


# Perform k-Fold validation
for train_index, test_index in skf.split(X_train_scaled, y_train):
    X_train_fold, X_test_fold = X_train_scaled[train_index], X_train_scaled[test_index]
    y_train_fold, y_test_fold = y_train[train_index], y_train[test_index]

    early_stopping = callbacks.EarlyStopping(
        patience=10,
        min_delta=0.0005,
        restore_best_weights=True,
    )

    # Train the model with class weight balancing
    train_history = NN_model.fit(
        X_train_fold, y_train_fold,
        validation_data=(X_test_fold, y_test_fold),
        batch_size=4,
        epochs=200,
        callbacks=[early_stopping],
        class_weight=class_weight
    )

    # Make predictions on the test set
    pred = NN_model.predict(X_test_fold)
    y_pred = [round(i[0]) for i in pred]

    # Calculate and store accuracy
    accuracy = accuracy_score(y_test_fold, y_pred)
    accuracies.append(accuracy)

    # Generate confusion matrix for this fold and store it
    cm_nn = confusion_matrix(y_test_fold, y_pred)
    confusion_matrices.append(cm_nn)

# Calculate the average accuracy across all folds
average_accuracy = sum(accuracies) / len(accuracies)
print('Average Neural Network accuracy with 3-Fold Validation:', round(average_accuracy, 3))
```

```
# Calculate the average confusion matrix across all folds
average_cm = sum(confusion_matrices) / len(confusion_matrices)
ax = plt.subplot()
sns.heatmap(average_cm, annot=True, ax=ax, fmt='g', cmap='Greens')

# Labels, title, and ticks
ax.set_xlabel('Predicted labels')
ax.set_ylabel('True labels')
ax.set_title('Average Neural Network Confusion Matrix for 5-Fold Validation With Weight balancing')

plt.show()
```

```
# Precision, Recall, and F1 Score for class 0
precision_class_0 = precision_score(y_test_fold, y_pred, pos_label=0)
recall_class_0 = recall_score(y_test_fold, y_pred, pos_label=0)
f1_class_0 = f1_score(y_test_fold, y_pred, pos_label=0)

# Precision, Recall, and F1 Score for class 1
precision_class_1 = precision_score(y_test_fold, y_pred, pos_label=1)
recall_class_1 = recall_score(y_test_fold, y_pred, pos_label=1)
f1_class_1 = f1_score(y_test_fold, y_pred, pos_label=1)

print(f'Class 0 Precision: {precision_class_0:.4f}, Recall:
{recall_class_0:.4f}, F1 Score: {f1_class_0:.4f}')
print(f'Class 1 Precision: {precision_class_1:.4f}, Recall:
{recall_class_1:.4f}, F1 Score: {f1_class_1:.4f}')
```




```
Class 0 Precision: 1.0000, Recall: 1.0000, F1 Score: 1.0000
Class 1 Precision: 1.0000, Recall: 1.0000, F1 Score: 1.0000
```

1. **Imports**:
   - Importing necessary modules from `keras`, `tensorflow`, `sklearn`, `seaborn`, and `matplotlib`.

2. **Define Class Weights**:
   - This calculates class weights for a binary classification problem, which is used to handle imbalanced datasets.
   - The weight for each class is calculated as the total number of samples divided by (2 times the number of samples in that class).

3. **Building the Neural Network**:
   - A simple feed-forward neural network is defined using Keras's Sequential API.
   - The network has three layers: two hidden layers with 16 and 8 neurons respectively (both using ReLU activation), and an output layer with a single neuron (using sigmoid activation for binary classification).
   - The model is then compiled with the binary cross-entropy loss function, the Adam optimiser, and it will track binary accuracy during training.

4. **k-Fold Cross Validation Setup**:
   - Stratified K-Fold is used, which means that each fold is made by preserving the percentage of samples for each class.
   - `k` is set to 5, meaning it'll create 5 different train-test sets for evaluation.

5. **Training Loop**:
   - The StratifiedKFold's `split` method provides indices to split the data into train and test sets.
   - The model is trained on the training fold and evaluated on the testing fold.
   - Early stopping is used, which will stop the training if the model hasn't improved by a delta of 0.0005 for 10 epochs. This helps in preventing overfitting.
   - Notably, the model training utilises the `class_weight` parameter to handle class imbalance.

6. **Predictions and Evaluation**:
   - After training on each fold, predictions are made on the test set.
   - The rounded predictions (`y_pred`) are compared with true labels to calculate the accuracy, which is then stored in the `accuracies` list.
   - A confusion matrix is generated for each fold and stored in the `confusion_matrices` list.

7. **Post-Processing**:
   - The average accuracy across all folds is calculated and printed.
   - An average confusion matrix is also computed by summing up the confusion matrices from each fold and dividing by the number of folds.

8. **Visualization**:
   - The average confusion matrix is visualised using Seaborn's heatmap.
   - This plot provides a visual representation of the model's performance across