A2.82 KNN With Weight Balancing

```
[ ] # Applyig Weight Balancing to address the class imbalance in the dataset

    from sklearn.neighbors import KNeighborsClassifier
    from sklearn.metrics import classification_report, confusion_matrix
    import seaborn as sns

    # Assuming you have already preprocessed the data and split it into X_train, X_test, y_train, and y_test.

    # 1. Class Weight Balancing:
    # Calculate the class weights to balance the classes
    class_weight = {0: len(y_train) / (2 * sum(y_train == 0)), 1: len(y_train) / (2 * sum(y_train == 1))}

    # 2. Create the KNN model with class weight balancing
    knn_weighted = KNeighborsClassifier(n_neighbors=3, weights='distance')

    # 3. Train the KNN model with class weight balancing
    knn_weighted.fit(X_train, y_train)

    # 4. Evaluate the KNN model on the test set
    print("KNN with Class Weight Balancing - Classification Report (Test Set) Results:")
    print(classification_report(y_test, knn_weighted.predict(X_test)))

    # 5. Evaluate the KNN model on the training set
    print("KNN with Class Weight Balancing - Classification Report (Training Set) Results:")
    print(classification_report(y_train, knn_weighted.predict(X_train)))

    # 6. Plot the confusion matrix for the KNN model with class weight balancing
    confmat_weighted = confusion_matrix(y_test, knn_weighted.predict(X_test))
    fig_weighted, ax_weighted = plt.subplots(figsize=(8, 8))
    g_weighted = sns.heatmap(confmat_weighted, annot=True, ax=ax_weighted, fmt='0.1f', cmap='Accent_r')
    g_weighted.set_yticklabels(g_weighted.get_yticklabels(), rotation=0, fontsize=12)
    g_weighted.set_xticklabels(g_weighted.get_xticklabels(), rotation=90, fontsize=12)
    ax_weighted.set_xlabel('Predicted labels')
    ax_weighted.set_ylabel('True labels')
    plt.title("Confusion Matrix for KNN with Class Weight Balancing")
    plt.show()
```

The code provided demonstrates the steps to use K-Nearest Neighbors (KNN) with class weight balancing to address the class imbalance in the dataset. Let's go step by step:

1. **Introduction**: The comment at the beginning mentions that the intent is to apply weight balancing to address class imbalance in the dataset. Class imbalance can often lead to biased predictions, favoring the majority class.

2. **Imports**:
   - `KNeighborsClassifier` from `sklearn.neighbors` is the KNN classifier.
   - Functions from `sklearn.metrics` are imported to evaluate the model's performance.
   - `seaborn` is imported for data visualization, specifically for plotting the confusion matrix.

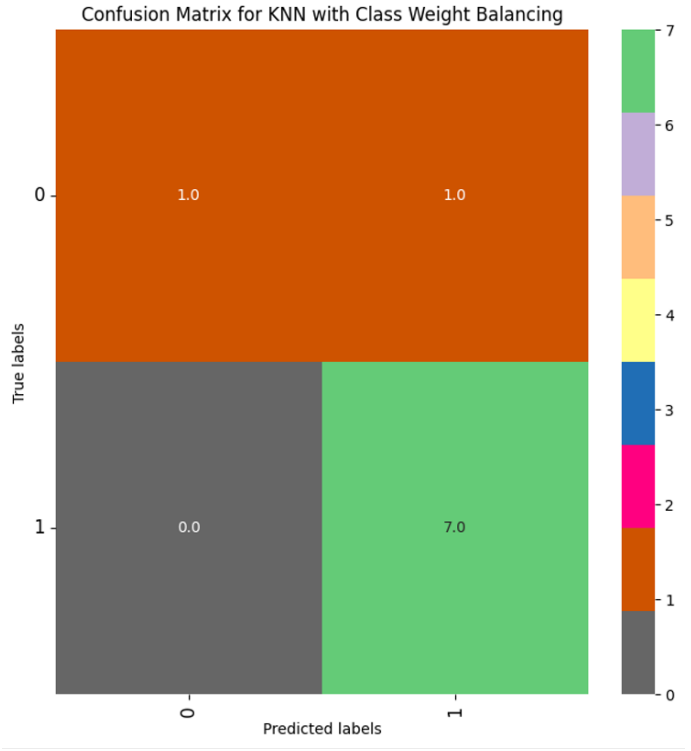
3. **Class Weight Calculation**:
   - `class_weight` is a dictionary that computes the class weights to balance the classes. The idea is to give more weight to the minority class and less weight to the majority class.
   - For class `0`, the weight is calculated as the total number of training samples divided by twice the number of class `0` samples.
   - For class `1`, the weight is calculated similarly.

4. **Creating and Training the KNN Model**:
   - A KNN model (`knn_weighted`) is initialised with `n_neighbors=3` indicating it will consider the three closest data points to make a prediction.
   - The `weights='distance'` parameter means that the closer neighbours of a query point will have a greater influence than neighbours that are further away. This is different from the default 'uniform' weights where each of the k neighbours has equal vote irrespective of distance.
   - Note: The `class_weight` dictionary we computed is not directly used in the KNN model, unlike some other classifiers where you can pass class weights directly. Instead, using `weights='distance'` is a way to let closer points influence the prediction more.

5. **Model Evaluation**:
   - The KNN model's performance is evaluated on both the test and training datasets using the `classification_report`. This report provides metrics like precision, recall, and F1-score which can help in understanding the model's performance.

6. **Plotting the Confusion Matrix**:
   - A confusion matrix for the predictions on the test set is plotted. This matrix gives a clear picture of true positives, true negatives, false positives, and false negatives.
   - The matrix is plotted using `seaborn's` heatmap function. The matrix is annotated, meaning each cell will display the count of samples.
   - Labels for the x and y axis are set, and the title indicates that this is the

```
KNN with Class Weight Balancing - Classification Report (Test Set):
              precision    recall  f1-score   support

           0       1.00      0.50      0.67         2
           1       0.88      1.00      0.93         7

    accuracy                           0.89         9
   macro avg       0.94      0.75      0.80         9
weighted avg       0.90      0.89      0.87         9

KNN with Class Weight Balancing - Classification Report (Training Set):
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         4
           1       1.00      1.00      1.00        16

    accuracy                           1.00        20
   macro avg       1.00      1.00      1.00        20
weighted avg       1.00      1.00      1.00        20
```



Classification reports for the KNN model with class weight balancing:
Test Set:
1. **Class 0**:
   - **Precision**: The model correctly predicted all samples of class `0` that it claimed (100%). In other words, there were no false positives for class `0`.
   - **Recall**: Out of all actual class `0` samples, the model correctly predicted only half of them (50%). This means there were false negatives.
   - **F1-score**: This score is the harmonic mean of precision and recall. A score of `0.67` indicates a balance between precision and recall, though skewed towards precision in this case.

2. **Class 1**:
   - **Precision**: The model correctly predicted 88% of samples that it claimed were class `1`.
   - **Recall**: The model correctly identified all of the actual class `1` samples (100%).
   - **F1-score**: A score of `0.93` indicates a very good balance between precision and recall for class `1`.

3. **Overall**:
   - **Accuracy**: Out of all the samples, 89% were correctly classified.
   - **Macro Avg**: The unweighted average of the metric (precision, recall, f1-score) for each class is around `0.75` to `0.94`.
   - **Weighted Avg**: The average of the metric (precision, recall, f1-score) for each class, weighted by the number of true instances in each class. Here, the values are around `0.87` to `0.90`, which is quite good.

Training Set:
1. **Class 0 & Class 1**:
   - **Precision, Recall, and F1-score**: All these metrics are 100% for both the classes. This means the model perfectly predicted every sample in the training set.

2. **Overall**:
   - **Accuracy**: The accuracy is `1.00` or 100%. This means that every single instance in the training set was correctly classified.
   - **Macro Avg & Weighted Avg**: All these metrics indicate perfect classification in the training set, with values being `1.00` for both classes.

Output Interpretation of training and test data:
1. The KNN model with class weight balancing performs extremely well on the training data, achieving a perfect score. This could be indicative of overfitting, as the model seems to have memorised the training data rather than generalised from it.
2. On the test data, the model performs reasonably well, especially for class `1`. However, there is some difficulty in correctly classifying all the samples of class `0`.
3. While the precision is high for both classes in the test set, the recall for class `0` indicates there's room for improvement in capturing all the positive samples of this