A weighted approach to the K-Nearest Neighbors

Daniel Gebran
Electrical and Computer Engineering
California State University, Fresno
Fresno, CA

Caleb Dulay
Electrical and Computer Engineering
California State University, Fresno
Fresno, CA

Henry Delgado
Electrical and Computer Engineering
California State University, Fresno
Fresno, CA

***Abstract*—This report represents a comparative analysis of the K-Nearest Neighbors (KNN) and Weighted K-Nearest Neighbors (WKNN). KNN is a non-parametric and versatile classification algorithm widely used in machine learning and pattern recognition. The WKNN algorithm extends the capabilities of KNN by incorporating weighted distances to improve classification accuracy. This report provides an overview of both algorithms, their implementation details, and a comprehensive evaluation of their performance using various datasets. The experimental results show the advantages and limitations of each algorithm and provide insights into their practical applications.**

I. Introduction

The K-Nearest Neighbor (KNN) algorithm is a popular instance-based learning method used for classification and regression tasks. It is a non-parametric approach that makes predictions based on the majority vote of the K nearest neighbors to a given data point. KNN has been widely utilized in various domains, including image recognition, text mining, and bioinformatics. However, KNN does not consider the relative importance of neighboring points, which may lead to suboptimal results in certain scenarios. To address this limitation, the Weighted K-Nearest Neighbor (WKNN) algorithm introduces weighted distances to assign different weights to neighboring points based on their relevance.

II. Background

The k-nearest neighbors algorithm, also referred to as KNN or k-NN, is a supervised learning classifier that falls under the non-parametric category. It leverages proximity to make predictions or classifications regarding the grouping of an individual data point. Although it can be employed for regression problems as well, it is commonly used as a classification algorithm. The fundamental assumption underlying KNN is that similar data points tend to be located in close proximity to each other [1].

In the case of classification tasks, a class label is assigned based on a majority vote. This means that the label most frequently represented among the neighboring data points is utilized. While this process is technically referred to as "plurality voting," the term "majority vote" is more commonly used in literature. It is important to note the distinction between these terminologies. "Majority voting" technically requires a majority of more than 50%, which is suitable when dealing with only two categories. However, when there are multiple classes, such as four categories, it is not necessary to have over 50% of the votes to assign a class label. Instead, a class can be assigned with a vote exceeding 25%. [1]

Weighted kNN is a variant of the k-nearest neighbors algorithm, designed to address certain limitations. One of the key challenges in the original kNN algorithm is selecting an appropriate value for the hyperparameter k. If k is set too small, the algorithm becomes highly sensitive to outliers, potentially leading to less reliable classifications. Conversely, if k is set too large, the neighborhood may encompass too many points from other classes, affecting the accuracy of predictions.

Another concern lies in the method of combining class labels during the classification process. The conventional approach is to use a majority vote, where the class label with the highest frequency among the nearest neighbors is assigned. However, this approach can be problematic when the nearest neighbors have varying distances, and the closest neighbors provide more reliable indications of the object's class. In such cases, a straightforward majority vote may not accurately capture the true class of the object, highlighting the need for an alternative approach. [3]

In this report, we provide a comprehensive analysis of both the KNN and WKNN algorithms. We discuss the fundamental concepts of KNN and its implementation details. We then introduce the WKNN algorithm and explain how it addresses the limitations of KNN through the incorporation of weighted distances. Through this analysis, we aim to provide insights into the strengths, weaknesses, and practical applications of both algorithms.

III. Evaluation of Experiment

This section presents the experimental evaluation of both the KNN and WKNN algorithms, including the evaluation metrics used to assess their performance.

A. Confusion Matrix

In order to obtain those metrics, a Confusion Matrix is needed first to group the classification results into four categories: True Positive (TP), True Negative (NP), False Positive (FP), and False Negative (FN).

True Positive refers to the number of samples when both the actual and predicted values are “1”, and True Negative refers to the one when both values are “0”. On the other hand, False Positive refers to the number of samples when the actual value is “0” but the predicted value is “1”, and False Negative refers to the one when the actual value “1” but the predicted value is “0” [4]. Fig.1 shows the typical layout for a confusion matrix.

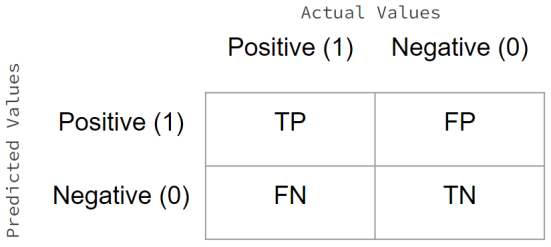

Fig.1. Confusion Matrix

Once the confusion matrix is populated, the counts for each category are used in the calculation of the following evaluation metrics.

B. Accuracy

Accuracy measures the overall correctness of the classification results. It calculates the ratio of correctly classified instances to the total number of instances [4]:

$$\frac{Num.\ of\ correct\ predictions}{Total\ num.\ of\ predictions}$$

A high accuracy indicates a good performance, but it may not be sufficient when dealing with imbalanced datasets.

C. Precision

Precision measures the proportion of correctly classified positive instances out of all instances predicted as positive. It focuses on the correctness of positive predictions and helps evaluate the algorithm's ability to avoid false positive errors. Precision is calculated as the ratio of true positives to the sum of true positives and false positives [4]:

$$\frac{True\ Positives\ (TP)}{True\ Positives\ (TP) + False\ Positives\ (FP)}$$

D. Recall

Recall, also known as sensitivity or true positive rate, measures the proportion of correctly classified positive instances out of all actual positive instances. It focuses on the algorithm's ability to identify all positive instances and avoid false negative errors. Recall is calculated as the ratio of true positives to the sum of true positives and false negatives [4]:

$$\frac{True\ Positives\ (TP)}{True\ Positives\ (TP) + False\ Negatives\ (FN)}$$

E. F1 Score

The F1 score is the harmonic mean of precision and recall and provides a balanced measure of the algorithm's performance. It combines the precision and recall values into a single metric and is particularly useful when dealing with imbalanced datasets. The F1 score is calculated as 2 times the product of precision and recall, divided by the sum of precision and recall [4]:

$$\frac{2 \cdot Precision \cdot Recall}{Precision + Recall}$$

IV. Proposed Method

As said earlier, the KNN algorithm can run into some limitations that could affect its performance, such as the choice of the hyperparameter k and the combination of class labels using the majority vote approach. To counteract them, a modified version of KNN can be used by assigning weights to the nearest k neighbors, which gives an alternative to the outcome selection process of the prediction. It is called the Weighted K-Nearest Neighbors algorithm (WKNN).

The reasoning behind Weighted KNN is the following. The closer a point is to the query point (the one to be predicted), the higher its weight is going to be. In contrast, the further a point is to the query point, the lower its weight is going to be. In this way, the weight of a point is determined by its distance from the query point and is usually computed by the inverse distance function (1/distance) [2], which confirms the weight/distance scale since the function result is smaller as distance grows and bigger as distance decays.

Given the information we have, we proposed to compare both KNN and Weighted KNN implementations by evaluating their prediction performance on a same dataset. The results we get would allow us to determine whether or not Weighted KNN brings accuracy, precision, recall, and therefore F1 score improvements to our prediction model. Our chosen dataset for this comparison has 400 data points of people with given gender (Male or Female), age (18-60), salary (15,000-150,000), and whether they would purchase an iPhone or not (0 or 1) [6]. Using the KNN implementations, the gender, age, and salary features (X variables) would be determining the target (Y variable), whether or not a person would buy an iPhone.

V. Experimentation

Our experiment was realized on Visual Studio Code on a computer running Windows 11 and housing an Intel Core i7 processor along with 16GB of RAM. Our programming language of choice was C++.

The first step in our experiment was to implement a `readCSV()` function that can read our dataset file and insert each person's data into a vector of "Person" objects, with each object including the gender, age, salary, and iPhone purchase outcome.

Once that was done, the two KNN algorithms were implemented in code as the `predict()` (basic KNN) and `weightedKNN()` functions. Both implementations were grouped within a single "KNNClassifier" class, which includes our training data from the dataset as a vector of "Person" objects and the k value as an integer. In addition, both algorithms had an underlying Euclidean Distance function that had to be implemented as well, which was tweaked to account for the multiple features of our dataset.

Next, we implemented a `trainTestSplit()` function that splits our dataset into a training set and testing set. The reason why we do this is to have part of the dataset used for training the model and have another part used for predictions and evaluating the model's performance. In other words, the training data is used to learn the relationship between the features and the target, and the test data is used to test the model by making predictions on it, which are then compared to the actual target values [5]. Since we need a decent amount of data to refer to when making predictions, we account 80% of the dataset for the training set as a convention, leaving 20% for the testing set.

Furthermore, a function that builds a confusion matrix, based on the predicted and actual values, was also implemented within the same class, along with other functions that calculate the accuracy, precision, recall, and F1 scores for the given prediction results, including the resulting confusion matrix.

Having all our functions ready to be tested, we conducted our experiment following the steps below. The full commented code can be found in the Appendix section at the end of this paper.

A. Instantiate KNN model and prepare the data

First of all, we initialize a "knn" object from the "KNNClassifier" class with a *k* value of 5. This value was

chosen for its optimal intensity and the fact that it's odd. Next, we call the `readCSV()` function to read the iPhone purchase dataset labeled as "iphone_purchase_records.csv", then populate the feature variables and target variable into two separate vectors X and y.

Once the data is imported, we call the `trainTestSplit()` function to split our dataset into a training set ("X_train" and "y_train") and a testing set ("X_test" and "y_test"), from the populated "X" and "y" vectors.

B. *Make predictions on the testing set*

Using the training data stored in the "X_train" vector, we make predictions on the testing set using the basic KNN algorithm within the `predict()` function. For every prediction function call, the result is stored in a "y_pred" vector, which is going to be used later for model performance evaluation.

As prediction was done using basic KNN, repeat the same few steps from the previous paragraph but this time using the WKNN algorithm. The difference is that the result is stored in a "y_weighted_pred" vector, also planning to be used later for performance evaluation.

C. *Evaluate the performance of KNN model*

Having all the necessary functions for performance evaluation, we evaluate basic KNN then WKNN, and we proceed to compare the performance results.

The evaluation is done as follows. Since the accuracy score can be computed without the confusion matrix, we first call the `accuracyScore()` function for the basic KNN model, given its "y_test" testing set target vector and its "y_pred" prediction results vector. Next, we create the confusion matrix for the basic KNN model, given those same two vectors, and store it in memory. Once that's done, we have now obtained the TP, TN, FP, and FN values that would be used accordingly for the other metric calculations. Therefore, we then compute the precision and recall scores by calling the `precisionScore()` and `recallScore()` functions, which depend upon the confusion matrix category values that could be accessed using multi-dimensional vector indexing. And finally, the F1 score is then calculated using the precision and recall scores computed earlier. We proceed to print the confusion matrix along with the performance metric scores to the console.

After basic KNN is evaluated, we repeat the same steps for WKNN and print the results to the console.

VI. Results

Since the `trainTestSplit()` function splits the dataset randomly each time we run it, we decided to perform three trials and compare the performance results. We can see that from trials 1 and 3 in Fig. 2 and Fig. 4, weighted KNN outperformed and optimized our performance metrics. However, we can see in Fig. 3 that it is not the case for trial 2. After testing basic KNN and WKNN more than three times, we noticed that outperformance is sometimes possible for basic KNN over WKNN, but WKNN manages to beat its basic version more often.

In addition, when it comes to having an imbalanced dataset, we would prefer to use weighted KNN since there would be no bias as there would be in KNN. As a result, we can conclude that most of the time weighted KNN will outperform basic KNN.

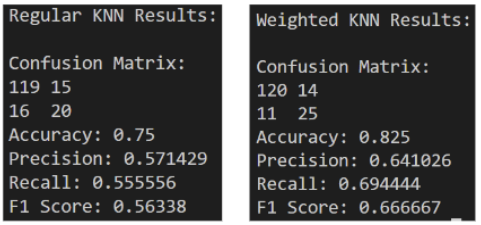

Fig.2. KNN vs WKNN (Trial 1)

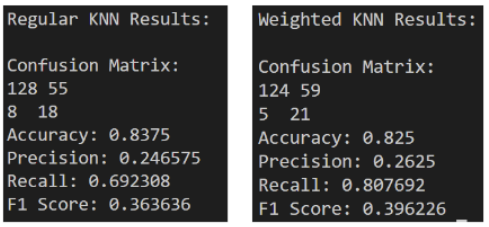

Fig.3. KNN vs WKNN (Trial 2)

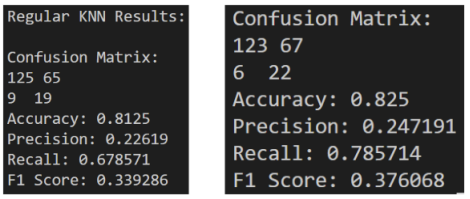

Fig.4. KNN vs WKNN (Trial 3)

VII. Analysis

The time complexity of both the K-Nearest Neighbors (KNN) algorithm and the Weighted K-Nearest Neighbors (WKNN) algorithm depends on several factors, including the number of training instances (N), the number of features (D), and the value of the hyperparameter K. The time and space complexity of this experiment was O(N*D).

The time complexity of weighted KNN vs basic KNN is generally the same, as the additional computations for weighing the contributions of neighbors is usually a constant factor and does not significantly impact the overall complexity.

VIII. Conclusion

In conclusion, this project dove into the implementation and optimization of the KNN algorithm, with a primary focus on enhancing key performance metrics such as recall, F1, accuracy, and precision scores through the incorporation of weighted KNN. The standard KNN algorithm, when dealing with imbalanced datasets, may not have been as accurate as we wanted it to be and as it is when dealing with balanced sets. Recognizing this limitation, we thought that weighted KNN proved to be a good way to optimize the performance metrics. The careful consideration of these metrics, as well as confusion matrix, ensures a well-rounded evaluation of the model's performance in real-world applications. And since time and space complexities have a minor difference between both implementations, weighted KNN is still a better choice over basic KNN. Overall, the integration of weighted KNN into the project's framework

emerged as a powerful enhancement, successfully addressing the challenge posed for KNN by imbalanced datasets and we can see that from the results that we've obtained, we could say that our hypothesis/objective was achieved as we were able to use weighted KNN to optimize accuracy, precision, F1 score, and recall.

References

[1] "What is the K-nearest neighbors algorithm?," IBM, https://www.ibm.com/topics/knn (accessed Dec. 10, 2023).

[2] kanakalathav99, "Weighted K-NN," GeeksforGeeks, https://www.geeksforgeeks.org/weighted-k-nn/ (accessed Nov. 8, 2023).

[3] S. Raschka, "Machine Learning," https://sebastianraschka.com/, https://sebastianraschka.com/pdf/lecture-notes/stat479fs18/02_knn_notes.pdf (accessed 2023).

[4] M. Amer, "Classification evaluation metrics: Accuracy, precision, recall, and F1 visually explained," Context by Cohere, https://txt.cohere.com/classification-eval-metrics/ (accessed Dec. 10, 2023).

[5] A. Shafi, "K-Nearest Neighbors (KNN) classification with scikit-learn," DataCamp, https://www.datacamp.com/tutorial/k-nearest-neighbor-classification-scikit-learn (accessed Dec. 10, 2023).

[6] P. Borhade, "KNN algorithms," Kaggle, https://www.kaggle.com/datasets/piyushborhade/knn-algorithms (accessed Dec. 11, 2023).

Appendix

In this section, the full code of our proposed method can be found below:

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <string>
#include <cmath>
#include <algorithm>
#include <random>
using namespace std;

struct Person {
    string gender;
    int age;
    int salary;
    int purchaseIphone;
};

class KNNClassifier {
    private:
        vector<Person> trainingData;
        int k;
         double euclideanDistance(const Person& p1, const Person& p2);
    public:
        KNNClassifier(int k): k(k) {};

            void trainTestSplit(const vector<Person>& X, const vector<int>& y,
                                double test_size,
                                vector<Person>& X_train, vector<int>& y_train,
                                vector<Person>& X_test, vector<int>& y_test);
         int predict(const vector<Person>& dataset, const Person& newPerson);
        vector<Person> readCSV(const string& filename);
          int weightedKNN(const vector<Person>& dataset, const Person& newPerson);
         vector<vector<int>> confusionMatrix(vector<int>& y_test, vector<int>& y_pred);
              double accuracyScore(vector<int>& y_test, vector<int>& y_pred);
             double precisionScore(int truePositive, int falsePositive);
              double recallScore(int truePositive, int falseNegative);
        double F1Score(double precision, double recall);
        void setK(int newK);

};

double KNNClassifier::euclideanDistance(const Person& p1, const Person& p2) {
        // Euclidean distance calculation with gender included
    int genderFactor = (p1.gender == p2.gender) ? 0 : 1;
     return sqrt(pow(p1.age - p2.age, 2) + pow(p1.salary - p2.salary, 2) + pow(genderFactor, 2));
}

void KNNClassifier::trainTestSplit(const vector<Person>& X, const vector<int>& y,
                                   double test_size,
                                           vector<Person>& X_train, vector<int>& y_train,
                                           vector<Person>& X_test, vector<int>& y_test)
{
    // Combine X and y into a single dataset
    vector<pair<Person, int>> dataset;
    for (size_t i = 0; i < X.size(); i++) {
        dataset.push_back({X[i], y[i]});
    }

    // Shuffle the dataset randomly
    random_device rd;
    default_random_engine rng(rd());
    shuffle(dataset.begin(), dataset.end(), rng);

     // Calculate the split index based on the test_size ratio
     size_t split_index = static_cast<size_t>(test_size * dataset.size());

    // Clear the training and testing set variables
    X_train.clear();
    y_train.clear();
    X_test.clear();
    y_test.clear();
```

```
    // Split the dataset into training and testing sets
    for (size_t i = 0; i < dataset.size(); i++) {
        if (i < split_index) {
            X_train.push_back(dataset[i].first);
            y_train.push_back(dataset[i].second);
        } else {
            X_test.push_back(dataset[i].first);
            y_test.push_back(dataset[i].second);
        }
    }
}


int KNNClassifier::predict(const vector<Person>& dataset, const Person& newPerson) {
     // Calculate distances between the new person and all existing persons in the dataset
    vector<pair<double, int>> distances;
    for (size_t i = 0; i < dataset.size(); ++i) {
          double distance = euclideanDistance(newPerson, dataset[i]);
        distances.emplace_back(distance, i);
    }

    // Sort distances in ascending order
    sort(distances.begin(), distances.end());

     // Count the number of purchases among the k nearest neighbors
    int purchaseCount = 0;
    for (int i = 0; i < k; ++i) {
        int neighborIndex = distances[i].second;
                                    purchaseCount     += dataset[neighborIndex].purchaseIphone;
    }

    // Predict purchase based on majority vote
    return (purchaseCount > k / 2) ? 1 : 0;
}


vector<Person>    KNNClassifier::readCSV(const    string& filename) {
    // vector<Person> data;
    ifstream file(filename);

    if (!file.is_open()) {
          cerr << "Error opening file: " << filename << endl;
         return trainingData;  // Return an empty vector in case of an error
    }

    // Read the header line to skip it
    string line;
    getline(file, line);

    // Read the data
    while (getline(file, line)) {
        istringstream iss(line);
        string gender;
        string ageStr, salaryStr, purchaseStr;

                    // Assuming the CSV structure is "Gender,Age,Salary,Purchase Iphone"
        getline(iss, gender, ',');
        getline(iss, ageStr, ',');
        getline(iss, salaryStr, ',');
        getline(iss, purchaseStr, ',');

        // Convert strings to the appropriate types
        Person person;
        person.gender = gender;
        person.age = stoi(ageStr);
        person.salary = stoi(salaryStr);
        person.purchaseIphone = stoi(purchaseStr);

        // Add the person to the vector
        trainingData.push_back(person);
    }

    file.close();
    return trainingData;
}


int    KNNClassifier::weightedKNN(const    vector<Person>& dataset, const Person& newPerson) {
     // Calculate distances between the new person and all existing persons in the dataset
    vector<pair<double, int>> distances;
    for (size_t i = 0; i < dataset.size(); ++i) {
          double distance = euclideanDistance(newPerson, dataset[i]);
        distances.emplace_back(distance, i);
    }

    // Sort distances in ascending order
    sort(distances.begin(), distances.end());

    // Consider the first k elements and two groups
    double freq1 = 0;
    double freq2 = 0;
     // Compute weighing function and increment for each frequency
    for (int i = 0; i < k; i++) {
        int neighborIndex = distances[i].second;
        if (dataset[neighborIndex].purchaseIphone == 0) {
            freq1 += double(1 / distances[i].first);
        }
        else if (dataset[neighborIndex].purchaseIphone == 1) {
            freq2 += double(1 / distances[i].first);
        }
```

```
    }
    return (freq1 > freq2 ? 0 : 1);
}

double KNNClassifier::accuracyScore(vector<int>& y_test, vector<int>& y_pred) {
    int correctPredictions = 0;
    size_t numPredictions = y_pred.size();
    for (int i = 0; i < y_pred.size(); i++) {
        if (y_pred[i] == y_test[i]) {
            correctPredictions++;
        }
    }
        double accuracy = double(correctPredictions) / double(numPredictions);
    return accuracy;
}

vector<vector<int>> KNNClassifier::confusionMatrix(vector<int>& y_test, vector<int>& y_pred) {
    int tp, tn, fp, fn = 0;

    for (int i = 0; i < y_pred.size(); i++) {
         // True Positive (TP): when both the actual and predicted values are 1.
        if (y_test[i] == 1 && y_pred[i] == 1) {
            tp++;
        }
         // True Negative (TN): when both the actual and predicted values are 0.
        else if (y_test[i] == 0 && y_pred[i] == 0) {
            tn++;
        }
         // False Positive (FP): when the actual value is 0 but the predicted value is 1.
        else if (y_test[i] == 0 && y_pred[i] == 1) {
            fp++;
        }
         // False Negative (FN): when the actual value is 1 but the predicted value is 0.
        else if (y_test[i] == 1 && y_pred[i] == 0) {
            fn++;
        }
    }

    // Populate the confusion matrix
    vector<vector<int>> confusionMatrix = {
        {tn, fp},
        {fn, tp}
    };

    return confusionMatrix;
}

double KNNClassifier::precisionScore(int truePositive, int falsePositive) {
         double precision = double(truePositive) / double(truePositive + falsePositive);
    return precision;
}

double KNNClassifier::recallScore(int truePositive, int falseNegative) {
          double recall = double(truePositive) / double(truePositive + falseNegative);
    return recall;
}

double KNNClassifier::F1Score(double precision, double recall) {
       double F1 = double(2 * precision * recall) / double(precision + recall);
    return F1;
}

void KNNClassifier::setK(int newK) {
    if (newK > 0) {
        k = newK;
    }
    else {
         cout << "Error: k must be a positive integer." << endl;
    }
}

int main() {
    // Instantiate KNN model with k = 5
    KNNClassifier knn(5);

    // Define the dataset
                            vector<Person> X = knn.readCSV("iphone_purchase_records.csv");
    vector<int> y;
    for (int i = 0; i < X.size(); i++) {
        y.push_back(X[i].purchaseIphone);
    }

    double test_size = 0.8; // 80% training, 20% testing

    vector<Person> X_train;
    vector<int> y_train;
    vector<Person> X_test;
    vector<int> y_test;

     knn.trainTestSplit(X, y, test_size, X_train, y_train, X_test, y_test);

    // Make predictions on the test dataset
    vector<int> y_pred;
    for (int i = 0; i < X_test.size(); i++) {
        int prediction = knn.predict(X_train, X_test[i]);
        y_pred.push_back(prediction);
```

```
    }


    // Evaluate Prediction Accuracy of Standard KNN
    double accuracy = knn.accuracyScore(y_test, y_pred);
                vector<vector<int>>    confusionMatrix    =
knn.confusionMatrix(y_test, y_pred);
                              double       precision       =
knn.precisionScore(confusionMatrix[1][1],
confusionMatrix[0][1]);
                               double       recall       =
knn.recallScore(confusionMatrix[1][1],
confusionMatrix[1][0]);
    double F1 = knn.F1Score(precision, recall);


    cout << "Regular KNN Results: " << endl << endl;
    cout << "Confusion Matrix: " << endl;
           cout   <<   confusionMatrix[0][0]   <<   "   "   <<
confusionMatrix[0][1] << endl;
         cout   <<   confusionMatrix[1][0]   <<   "     "   <<
confusionMatrix[1][1] << endl;
    cout << "Accuracy: " << accuracy << endl;
    cout << "Precision: " << precision << endl;
    cout << "Recall: " << recall << endl;
    cout << "F1 Score: " << F1 << endl;
    cout << endl;


      // Use the Training and Testing sets to perform
Weighted KNN prediction
    vector<int> y_weighted_pred;
    for (int i = 0; i < X_test.size(); i++) {
                            int    weightedPredicition    =
knn.weightedKNN(X_train, X_test[i]);
        y_weighted_pred.push_back(weightedPredicition);
    }


    // Evaluate Prediction Accuracy of Weighted KNN
         double   w_accuracy   =   knn.accuracyScore(y_test,
y_weighted_pred);
              vector<vector<int>>    w_confusionMatrix    =
knn.confusionMatrix(y_test, y_weighted_pred);
                           double       w_precision       =
knn.precisionScore(w_confusionMatrix[1][1],
w_confusionMatrix[0][1]);
                               double       w_recall       =
knn.recallScore(w_confusionMatrix[1][1],
w_confusionMatrix[1][0]);
    double w_F1 = knn.F1Score(w_precision, w_recall);



    cout << "Weighted KNN Results: " << endl << endl;
    cout << "Confusion Matrix: " << endl;
          cout   <<   w_confusionMatrix[0][0]   <<   "   "   <<
w_confusionMatrix[0][1] << endl;
        cout   <<   w_confusionMatrix[1][0]   <<   "     "   <<
w_confusionMatrix[1][1] << endl;
    cout << "Accuracy: " << w_accuracy << endl;
    cout << "Precision: " << w_precision << endl;
    cout << "Recall: " << w_recall << endl;
    cout << "F1 Score: " << w_F1 << endl;


    return 0;
}
```