CSE 5160 - Project Report
KNN Classifier Implementation

Alyssa Wilcox
School of Computer Science and Engineering
California State University San Bernardino
San Bernardino, CA 92407
https://www.csusb.edu/cse

Abstract

This article examines the implementation of a k nearest neighbor (KNN) machine learning classifier. The goal is to describe the implementation process through the functions used in the implementation, explain how the training set builds the classifier, and display the results after applying the classifier on test instances. This classifier is built specifically to classify three varieties of seeds: Kama (1), Rosa (2), and Canadian (3) wheat seeds. Seven measurements were taken from these wheat seeds, including seed: area, perimeter, compactness, length of kernel, width of kernel, asymmetry coefficient, and length of kernel groove. These seven features, in addition to the classification of the wheat seed, form a training instance. Many training instances create the training set, which is used by the classifier to classify a new test instance. Although KNN may not be an efficient machine learning algorithm, it still can be useful to classify unknown test instances based on training data.

1 Description of implementation process

This section describes the implementation of the KNN classifier. Specifically, it describes the functions used in the implementation: what functions were used, what they do, and how they work together to build and implement the KNN classifier.

1.1 Main function - main()

The implementation of the classifier begins with a main function that serves as an entry point and driver of the program. It takes in no parameters and returns 0 upon program completion. This function first displays an introduction to the user, displaying what the program is. Afterwards, a do-while loop is used to allow the user to run the program multiple times. In this do-while loop, the classifier is implemented.

First, the classifier gets both the test instance from the user and the number of k neighbors to use when classifying. For the test instance, all features of the test instance should be double variables. For the value k, it should be a integer variable. To ensure valid input, an input validation function is called to validate the feature input for the test instance before storing it to a test instance vector. Additionally, another input validation function is called to validate the value of k.

Second, the classifier gets the training set data. The library <fstream>, specifically an input file stream object, is used to open a text file that contains the training set data. A two dimensional vector is used to contain the training set, and the function **getTrainingSet()** is called to fill this vector with training instances (Figure 1.1.1).

```
//Create vector to store training set
vector<vector<double>> training_set = getTrainingSet(infile);
```

(Figure 1.1.1)

Third, the classifier feature scales both the test instance and the training set through min-max normalization. To do this, the function **minMaxNormalization()** is called (Figure 1.1.2).

```
//min-max normalize training set and test instance
minMaxNormalization(training_set, test_instance);
```

(Figure 1.1.2)

Fourth, the classifier finds the Euclidean distances between the test instance and each training instance by calling the **distances()** function (Figure 1.1.3)

```
//Find euclidean distances between training instances and test instance
distances(training_set, test_instance);
```

(Figure 1.1.3)

Fifth, the classifier sorts the training set based on the Euclidean distances found in ascending order. The sort function **sort()** provided by the <algorithm> library is used (Figure 1.1.4)

```
//Sort the training set based on distances
sort(training_set.begin(), training_set.end(), sortcol);
```

(Figure 1.1.4)

Sixth, the classifier finds the output of the k nearest neighbors through the **KNNoutput()** function, storing these k outputs in a KNNoutputY vector (Figure 1.1.5).

```
//Get output of nearest neighbors
vector<int> KNNoutputY = KNNoutput(training_set, k);
```

(Figure 1.1.5)

Seventh, the classifier classifies the test instance with the **classify()** function, saving the classification into an integer called prediction (Figure 1.1.6).

```
//Make the prediction
int prediction = classify(KNNoutputY);
```

(Figure 1.1.6)

Eighth, the classifier outputs the classification of the test instance to the user through a switch statement (Figure 1.1.7).

```
//Display prediction results
switch (prediction) {
case 0:
    cout << "Could not classify, please enter a different k";
    break;
case 1:
    cout << "The input has been classified as a 1: Kama wheat seed";
    break;
case 2:
    cout << "The input has been classified as a 2: Rosa wheat seed";
    break;
case 3:
    cout << "The input has been classified as a 3: Canadian wheat seed";
    break;
default:
    cout << "Something went wrong!";
}
```

(Figure 1.1.7)

1.2 Get the training set - getTrainingSet()

This function reads the training set text file, saving each training instance to a training instance vector. Each training instance vector is then added to a training set two dimensional vector. This function has one parameter *infile* which is an input file stream object passed by reference. Upon completion, the function returns a two dimensional training set vector.

The function begins by creating double variables for each feature type in a training instance. These variables are A (area), P, (perimeter), C (compactness), L (length of the kernel), W (width of the kernel), AC (asymmetry coefficient), and LG (length of the kernel groove). Next, a two dimensional training set vector is created to hold each training instance. A while loop is then used to read through the training set text file, reading all seven features of a training instance and saving those features in a training instance vector. That training instance vector is then pushed onto the training set vector. After reading through the entire training set text file, the function returns the training set vector (Figure 1.2.1).

```
vector<vector<double> > getTrainingSet(ifstream& infile) {
    double A, P, C, L, W, AC, LG, y;
    vector<vector<double>> training_set;
    while (!infile.eof()) {
        infile >> A >> P >> C >> L >> W >> AC >> LG >> y;
        vector<double> training_inst = { A, P, C, L, W, AC, LG, y };
        training_set.push_back(training_inst);
    }
    return training_set;
}
```

(Figure 1.2.1)

1.3 Feature scaling through min-max normalization - minMaxNormalization()

This function performs feature scaling through min-max normalization. Two parameters are passed: the *training set* two dimensional vector and the *test instance* vector. Both are passed by reference.

The first part of the function finds the minimum and maximum values for each feature in the training set and test instance. Two vectors are created to hold these minimum and maximum feature values: *min values* and *max values*. A while-loop is then used to traverse the training set and test instance feature by feature, for a total of n features. In one iteration of this loop, a double variable *max*, which keeps track of the maximum n feature value, is initialized to zero. A double variable *min*, which keeps track of the minimum n feature value, is initialized to the first training instance's n feature value. A for-loop then traverses the n feature of each training instance, for a total of m training instances. This loop compares the *max* and *min* values with the m training instance's n feature. The variables *max* and *min* are replaced if a new max and min value are found. The while-loop then checks the n feature of the test instance, replacing *max* and *min* if necessary. After all training instance's and the test instance's n feature have been traversed through, the *max* and *min* variables are pushed onto the *max*

values and *min values* vectors, respectively. The while-loop continues for every n feature (Figure 1.3.1).

```
void minMaxNormalization(vector<vector<double> >& training_set, vector<double>& test_instance) {
    vector<double> max_values;
    vector<double> min_values;
    int n = 0;
    while (n < 7) {
        double max = 0;
        double min = training_set[0][n];
        for (int m = 0; m < training_set.size(); m++) {
            if (max < training_set[m][n])
                max = training_set[m][n];
            if (min > training_set[m][n])
                min = training_set[m][n];
        }
        if (max < test_instance[n])
            max = test_instance[n];
        if (min > test_instance[n])
            min = test_instance[n];
        max_values.push_back(max);
        min_values.push_back(min);
        n++;
    }
```

(Figure 1.3.1)

The second part of this function performs the min-max normalization. A while-loop is used to traverse the training set and test instance feature by feature, for a total of n features. In one iteration of the while-loop, the n feature of every training instance and the test instance is then min-max normalized using the n feature's minimum and maximum feature values found. The min-max normalization formula is used to calculate these values (Figure 1.3.2). The while-loop then continues for the remaining n features (Figure 1.3.3).

$$\frac{feature\ X\ value\ - \min(X)}{\max(x) - \min(x)}$$

(Figure 1.3.2)

```
n = 0;
while (n < 7) {
    for (int m = 0; m < training_set.size(); m++) {
        training_set[m][n] = ((training_set[m][n] - min_values[n]) / (max_values[n] - min_values[n]));
    }
    test_instance[n] = ((test_instance[n] - min_values[n]) / (max_values[n] - min_values[n]));
    n++;
}
```

(Figure 1.3.3)

1.4 Find Euclidean distances - distances()

This function finds the Euclidean distances, using the formula in Figure 1.4.1, between the test instance and each training instance. Two parameters are passed: the *training set* two dimensional vector, passed by reference, and the *test instance* vector, passed by constant reference.

$$d(\vec{p}, \vec{q}) = \sqrt{\sum_{i=0}^{n} (p_i - q_i)^2}$$

(Figure 1.4.1)

The function uses an outer for-loop to traverse through all m training instances. In this loop, a double variable *sum* is declared and initialized to zero. An inner for-loop is then used to traverse through n

features of the m training instance. This inner for-loop keeps track of the *sum* of the test instance's n feature minus the m training instance's n feature, squared. After the inner for-loop finishes and the *sum* value is found, the square root of that sum is added onto the back of the m training instance vector. The outer for-loop continues for the remaining m training instances (Figure 1.4.2).

```
void distances(vector<vector<double> >& training_set, const vector<double>& test_instance) {
    for (int m = 0; m < training_set.size(); m++) {
        double sum = 0;
        for (int n = 0; n < training_set[m].size() - 1; n++) {
            sum = sum + pow(test_instance[n] - training_set[m][n], 2);
        }
        training_set[m].push_back(sqrt(sum));
    }
}
```

(Figure 1.4.2)

1.5 Sort the training set based on distances - sort()

In **main()**, the sort function provided by the library <algorithm> is used to sort the training set based on distances in ascending order. This sort function uses a driver function called **sortcol()** (figure 1.5.1) to sort the training set based on ascending distances.

```
bool sortcol(const vector<double>& v1, const vector<double>& v2) {
    return v1[8] < v2[8];
}
```

(Figure 1.5.1)

1.6 Get the output of the k nearest neighbors - KNNoutput()

This part of the classifier finds the output of the k nearest neighbors in the sorted training set. Two parameters are passed: the *training set* two dimensional vector, passed by constant reference, and the *k* integer value.

The function begins by creating an integer *output* vector. A for-loop is then used to get the k nearest neighbor's output values, which are the output index values of the first k training instances in the sorted training set. The function then returns the output vector with the classifications of the k nearest neighbors (Figure 1.6.1).

```
vector<int> KNNoutput(const vector<vector<double> >& training_set, const int k) {
    vector<int> output;
    for (int m = 0; m < k; m++) {
        output.push_back(training_set[m][7]);
    }
    return output;
}
```

(Figure 1.6.1)

1.7 Classify the test instance - classify()

This function classifies the test instance based on the output of the k nearest neighbors. One parameter is passed: the *KNNoutput* vector passed by constant reference.

The function begins by declaring three integer values: *one*, *two*, and *three*. All are initialized to zero and are used to track how many output types there are in the *KNNoutput* vector. A for-loop is then used to count how many of each output type there are. Specifically, it counts how many 1s, 2s, and 3s there are in the *KNNoutput* vector. Finally, the function returns an integer value based on the majority output type (Figure 1.7.1).

```
int classify(const vector<int>& KNNoutput) {
    int one = 0;
    int two = 0;
    int three = 0;
    for (int i = 0; i < KNNoutput.size(); i++) {
        if (KNNoutput[i] == 1)
            one++;
        else if (KNNoutput[i] == 2)
            two++;
        else if (KNNoutput[i] == 3)
            three++;
    }
    if ((one > two) && (one > three)) {
        return 1;
    }
    else if ((two > one) && (two > three)) {
        return 2;
    }
    else if ((three > one) && (three > two)) {
        return 3;
    }
    else {
        return 0;
    }
}
```

(Figure 1.7.1)

2 Using the training set to build the classifier

This KNN classifier heavily relies on the training set data to classify test instances.

2.1 Training set used

The training set used with this classifier corresponds to 180 training instances of measurements and classification of a wheat seed. The data was provided by UCI's Machine Learning Repository and can be found at `https://archive.ics.uci.edu/ml/datasets/seeds`

2.2 Training instance features

Each training instance has seven features. Each feature corresponds to some measurement taken from three varieties of wheat seeds. The seven features are: Area A, Perimeter P, Compactness C, Length of Kernel L, Width of Kernel W, Asymmetry Coefficient AC, and Length of Kernel Groove LG

2.3 Training instance classifications

Each training instance is classified as one of three varieties of wheat seeds:

- Kama wheat seed, classified as a 1
- Rosa wheat seed, classified as a 2
- Canadian wheat seed, classified as a 3

2.4 How the training set built the classifier

The classifier works by calculating the Euclidean distances between the test instance and every training instance. The classifications of the k nearest training instances are then used to classify the test instance. The training set does not explicitly build a model for the classifier.

3 Applying the classifier on test instances

Test instances were used to determine the accuracy of the KNN classifier.

3.1 Test set data

The original data set consists of 210 training instances. Thirty of them were used as a test set, while the remaining 180 became the training set. The thirty test instances were chosen at random, with ten of them corresponding to Kama wheat seeds, ten of them corresponding to Rosa wheat seeds, and ten of them corresponding to Canadian wheat seeds.

3.2 Test results - Test error

To find test error, the KNN classifier was tested by using the thirty instance long test set. Four different values of k were used: 5, 10, 15, and 20 nearest neighbors. The results of this testing are represented in Figure 3.2.1. Within the figure, TI = Test Instance and corresponds to the test instance number within the test set. RO = Real Output and corresponds to the real classification of the test instance. PO = Predicted Output and corresponds to the classification found by the KNN classifier. Test instances that were correctly predicted are marked in green while test instances that were incorrectly predicted are marked in red.

k = 5			k = 10			k = 15			k = 20		
TI	RO	PO	TI	RO	PO	TI	RO	PO	TI	RO	PO
1	1	1	1	1	1	1	1	1	1	1	1
2	1	1	2	1	1	2	1	1	2	1	1
3	1	1	3	1	1	3	1	1	3	1	1
4	1	1	4	1	1	4	1	1	4	1	1
5	1	1	5	1	1	5	1	1	5	1	1
6	1	2	6	1	2	6	1	2	6	1	2
7	1	3	7	1	3	7	1	3	7	1	3
8	1	1	8	1	1	8	1	1	8	1	1
9	1	1	9	1	1	9	1	1	9	1	1
10	1	1	10	1	3	10	1	1	10	1	3
11	2	2	11	2	2	11	2	2	11	2	2
12	2	2	12	2	2	12	2	2	12	2	2
13	2	2	13	2	2	13	2	2	13	2	2
14	2	2	14	2	2	14	2	2	14	2	2
15	2	2	15	2	2	15	2	2	15	2	2
16	2	2	16	2	2	16	2	2	16	2	2
17	2	1	17	2	1	17	2	1	17	2	1
18	2	2	18	2	2	18	2	2	18	2	2
19	2	2	19	2	2	19	2	2	19	2	2
20	2	2	20	2	2	20	2	2	20	2	2
21	3	3	21	3	3	21	3	3	21	3	3
22	3	3	22	3	3	22	3	3	22	3	3
23	3	3	23	3	3	23	3	3	23	3	3
24	3	3	24	3	3	24	3	3	24	3	3
25	3	3	25	3	3	25	3	3	25	3	3
26	3	3	26	3	3	26	3	3	26	3	3
27	3	3	27	3	3	27	3	3	27	3	3
28	3	3	28	3	3	28	3	3	28	3	3
29	3	3	29	3	3	29	3	3	29	3	3
30	3	3	30	3	3	30	3	3	30	3	3

(Figure 3.2.1)

3.3 Test error

The test error found through these four tests is represented in Table 1.

Table 1:

k value	# Classified Correctly	# Classified Incorrectly	Accuracy	Test Error
5	27	3	90%	10%
10	26	4	87%	13%
15	27	3	90%	10%
20	26	4	87%	13%

3.4 Test results - Training error

To find training error, the KNN classifier was tested by using 10 randomly selected training instances. Four different values of k were used: 5, 10, 15, and 20 nearest neighbors. The results of the tests are represented in Figure 3.3.1. Within the figure, TI = Test Instance and corresponds to the test instance number within the test set. RO = Real Output and corresponds to the real classification of the test instance. PO = Predicted Output and corresponds to the classification found by the KNN classifier. Test instances that were correctly predicted are marked in green while test instances that were incorrectly predicted are marked in red.

k = 5			k = 10			k = 15			k = 20		
TI	RO	PO	TI	RO	PO	TI	RO	PO	TI	RO	PO
1	1	1	1	1	1	1	1	1	1	1	1
2	1	1	2	1	1	2	1	1	2	1	1
3	1	1	3	1	1	3	1	1	3	1	1
4	1	1	4	1	1	4	1	1	4	1	1
5	2	2	5	2	2	5	2	2	5	2	2
6	2	2	6	2	2	6	2	2	6	2	2
7	2	2	7	2	2	7	2	2	7	2	2
8	3	3	8	3	3	8	3	3	8	3	3
9	3	3	9	3	3	9	3	3	9	3	3
10	3	3	10	3	3	10	3	3	10	3	3

(Figure 3.3.1)

3.5 Test error

The training error found through these four tests is represented in Table 2.

Table 2:

k value	# Classified Correctly	# Classified Incorrectly	Accuracy	Training Error
5	10	0	100%	0%
10	10	0	100%	0%
15	10	0	100%	0%
20	10	0	100%	0%

4 Conclusion

This k nearest neighbor classifier classifies three varieties of wheat seeds: Kama, Rosa, and Canadian. To do this, 180 training instances are used, each instance with seven features corresponding to different measurements of the wheat seed. Using a variety of functions, a KNN classifier is implemented. A main function drives the execution order of the program. A getTrainingSet() function fills a two dimensional training set vector by reading in training instances from a training set text file. The function minMaxNormalization() feature scales both the test instance and the training set. A distance() function finds the Euclidean distances between the test instance and every training instance. Through a sort() function provided by the <algorithm> library, the training set is sorted based on ascending distances. KNNoutput() then finds the outputs of the k nearest neighbors in the sorted training set. Finally, classify() counts the number of each output type and classifies the test instance. The implementation of this classifier proves to be relatively accurate with a low test and training error. Although KNN is not efficient, it is still an interesting application of machine learning to classify unknown instances based off of training instances.