Implementation of K-Nearest Neighbor (k-NN) Algorithm with C#

Implementation Link

https://github.com/sunnyhillfc/KNN

1. Introduction

K-nearest neighbour (k-NN) is one of the most popular machine learning algorithms due to its ease of implementation, low computational power requirements, and high level of classification accuracy. *(Javatpoint, 2021)*

The kNN technique, first demonstrated in 1951 by statisticians Evelyn Fix and Joseph Hodges is a method of classification (or regression) that forms object classifications based on the closest objects in the training set. *(Fix and Hodges, 1951)*

Given a dataset $D = [x_i, \ldots\ldots, y\ \ ]$

Where$s$ represents the total number of data points in D

andx_i represents a datapoint from $0 \ \rightarrow s$

and y_i is the target class for x_i

The k-NN algorithm will need to determine class y_i for each $x_i$using a function f. The algorithm outputs a class$y$ for each point (classes are either *a* or *b*) e.g

$$[(x1,\ b)\ , (x2,\ a), (\ x3, a)]$$

To do this, k-NN assigns a classification to each data point x_i according to a majority vote of its neighbours. The algorithm uses a positive integer k value usually < 10 to decide how many of the nearest data points vote on the classification of x_i.

For example,

if $k\ = \ 2$ and $xi\ = \ 12$

and has nearest neighbours $[(x2,\ b)\ (x4, b)]$, both with class b

data point xi is assigned a class of b

In the following report, the k-NN algorithm has been implemented using C# code to classify the well-known iris dataset using pre-determined class values.

2.0 k-NN Algorithm Implementation

The algorithm implementation consists of the following key methods

- **Main()** - entry point into the program, creates an instance of knn.cs
- **LoadData()** - a method for loading in testing and training files
- **Classify()** - method to store calculated distance between test data and training data. Also responsible for console output
- **EuclideanDistance()** - implementation of the Euclidean distance formula

Additionally, the following variables are used

List<double[]> trainingSetValues	The list that holds the training data values
List<string> trainingSetClasses	The list that holds classes for the training data
List<double[]> testSetValues	The list that holds the testing values
List<string> testSetClasses	The list that holds the testing classes, used to compute the accuracy
int K	K nearest neighbours number, in this implementation it is set to 5
enum DataType	Used to distinguish between testing and training data
Int lines	Used for console output to count how many lines from the dataset have been read in

2.1 Program Initiailaisation

The entry point to the program is **main()** in Program.cs
The program requires the following inputs

iris.dat	The well-known iris dataset
test.dat	Testing data, with 1 of 3 iris classes for each record assigned
k	K nearest neighbours number, in this implementation it is set to 5

```
6   namespace KNN
7   {
8       class Program
9       {
10          static void Main(string[] args)
11          {
12              KNN knn = new KNN();
13
14              knn.LoadData("iris.dat", KNN.DataType.TRAININGDATA);
15              knn.LoadData("test.dat", KNN.DataType.TESTDATA);
16              knn.Classify(5);
17              Console.ReadKey();
18          }
19      }
20  }
```

1. Line 12 - Creates a new instance of knn.cs, the object that contains all the code for this algorithm
2. Line 14 - Access the public **LoadData()** method, passing it the name of the training set file and specifying its enumeration
3. Line 15 - Access the public **LoadData()** method, passing it the name of the test set file and specifying its enumeration
4. Line 16 - Pass the k nearest neighbour number, in this case, 5 is used.
5. Line 17 - **Console.ReadKey()** is used to hold the console open

2.2 Data Input

To input training and testing data, the **LoadData()** method is used. The method takes two parameters - the physical file name as string *path* and a variable for the DataType enumeration called *dataType*

```
29          public void LoadData(string path, DataType dataType)
30          {
31              StreamReader file = new StreamReader(path);
32              string line;
33
34              this.lines = 0;
35
36              Console.WriteLine("[i] reading data from {0} ...", path);
```

1. Line 31 - Create a new instance of C# file reader object and pass it the path of the file
2. Line 34 - Start the line counter at 0
3. Line 36 - Output the file path to the console

```
38            while((line = file.ReadLine()) != null)
39            {
40                // as we have a CSV file basically, split the line at each ','
41                string[] splitLine = line.Split(',').ToArray();
42
43                // and add them to a list
44                List<string> lineItems = new List<string>(splitLine.Length);
45                lineItems.AddRange(splitLine);
46
47                // create an appropiate array to hold the doubles from this line
48                double[] lineDoubles = new double[lineItems.Count - 1];
49                // and a string holding the class
50                string lineClass = lineItems.ElementAt(lineItems.Count - 1);
51
52                for(int i = 0; i < lineItems.Count - 1; i++)    // last item is the set class
53                {
54                    // convert each item in the list to a double
55                    double val = Double.Parse(lineItems.ElementAt(i));
56                    lineDoubles[i] = val;
57                }
58
59                // finally, save them
60
61                if (dataType == DataType.TRAININGDATA)
62                {
63                    this.trainingSetValues.Add(lineDoubles);
64                    this.trainingSetClasses.Add(lineClass);
65                }
66                else if(dataType == DataType.TESTDATA)
67                {
68                    this.testSetValues.Add(lineDoubles);
69                    this.testSetClasses.Add(lineClass);
70                }
71                this.lines++;
72            }
73
74            Console.WriteLine("[+] done. read {0} lines.", this.lines);
75
76            file.Close();
77        }
```

4. Line 38 - While loop, run this while the file is being read in line by line
5. Line 41 - Split the current line from the .csv file at the comma and store it in an array
6. Lines 44-45 - Add the array to a string list of the same length
7. Line 50 - a string holding the class
8. Lines 52-57 - for loop, to convert each item in the list to a double, stored in the array created on line 48
9. Lines 61-70 - add the arrays to the global arrays based on the specified enum.
10. Line 71 - increment the line counter
11. Line 74 - output the total lines read to the console
12. Line 76 - close the file reader

2.3 Euclidian Distance

Euclidian Distance is an implementation of the Pythagorean theorem to find the distance between two data points. The distance between two points is the sum of the squares of its cartesian coordinates (Paul, 2020).

To calculate Euclidian Distance, the following formula will need to be implemented in code:

$$\sqrt{\sum_{i=1}^{k}(x_i - y_i)^2}$$

Where the coordinates $(x_i \ - \ y_i)$ refer to one data point x and its class$y$ And the sum of all values is applied a square root.

The Euclidean distance formula is used in the method **EuclideanDistance** in KNN.cs

```
131
132         private static double EuclideanDistance(double[] sampleOne, double[] sampleTwo)
133         {
134             double d = 0.0;
135
136             for(int i = 0; i < sampleOne.Length; i++)
137             {
138                 double temp = sampleOne[i] - sampleTwo[i];
139                 d += temp * temp;
140             }
141             return Math.Sqrt(d);
142         }
```

The code works as follows:

1. Line 132 - method declaration, the return type is a double and parameters are
2. Line 134 - declare a double with a 0 value, this will represent the sum of $(x_i - y_i)^2$
3. Line 136 - for loop conditioned on the length of the x containing array's length
4. Line 138 - $(x_i - y_i)$; subtraction between datapoint and class
5. Line 139 - implement the sum part of the formula, add temp value to itself and square root it by temp * temp
6. Line 141 - method return is set to **Math.Sqrt(d)** will return the calculated value (d) square rooted.

2.4 Classification

To train and test the algorithm, a dataset with know class values will be used. Hence, The machine learning process chosen is supervised learning.

Given a training dataset $S = \{[(x_i, y_i), \ldots\ldots, (x\ \ , y\ \ )]\}$

Where$x_i$represents a datapoint from $0 \ \rightarrow s$

and y_i is the class for x_i

A supervised learning algorithm uses the function $f\!: \ X \rightarrow Y$

Where X is the input value space
and Y is the output class space

The k-NN algorithm will need to determine class y_i for each x_i using a function f. To do this each neighbour of x_i up to k is assigned an equal voting power with neighbours further than the specified k having voting power of 0. The implementation of the function is partially handled by the **Classify()** method, with algorithmic calculations passed to the priorly discussed **EuclideanDistance()** method

```
79          public void Classify(int neighborsNumber)
80          {
81              this.K = neighborsNumber;
82
83              // create an array where we store the distance from our test data and the training data -> [0]
84              // plus the index of the training data element -> [1]
85              double[][] distances = new double[trainingSetValues.Count][];
86
87              double accuracy = 0;
88              double correct = 0, testNumber = 0;
89
90              for (int i = 0; i < trainingSetValues.Count; i++)
91                  distances[i] = new double[2];
92
93              Console.WriteLine("[i] classifying...");
```

1. Line 79-81 - method parameter is the k value, assigned to the KNN instance from user input earlier
2. Line 85 - array for the distances between training and test points
3. Lines 87 -88 - Variables used in the accuracy calculation

```
95              // start computing
96              for(var test = 0; test < this.testSetValues.Count; test++)
97              {
98                  Parallel.For(0, trainingSetValues.Count, index =>
99                      {
100                         var dist = EuclideanDistance(this.testSetValues[test], this.trainingSetValues[index]);
101                         distances[index][0] = dist;
102                         distances[index][1] = index;
103                     }
104                 );
105
106                 Console.WriteLine("[+] closest K={0} neighbors: ", this.K);
107
108                 // sort and select first K of them
109                 var sortedDistances = distances.AsParallel().OrderBy(t => t[0]).Take(this.K);
110
111                 string realClass = testSetClasses[test];
112
113                 // print and check the result
114                 foreach (var d in sortedDistances)
115                 {
116                     string predictedClass = trainingSetClasses[(int) d[1]];
117                     if (string.Equals(realClass, predictedClass) == true)
118                         correct++;
119                     testNumber++;
120                     Console.WriteLine("[>>>] test {0}: real class: {1} predicted class: {2}", test, realClass, predictedClass);
121                 }
122             }
123
124             Console.WriteLine();
125
126             // compute and print the accuracy
127             accuracy = (correct / testNumber) * 100;
128             Console.WriteLine("[i] accuracy: {0}%", accuracy);
129
130         }
```

1. Line 96 - For loop incremented on each test run, to run as many times as there are test values
2. Line 98 - Parallel for loop, this is to speed up processing by using multiple threads
3. Lines 99-103 - Iterate through the values, passing to the **EuclideanDistance()** method and storing the resulting distance in the parallel arrays

4. Line 106 - Output for each test run
5. Line 109 - sort the array
6. Line 111- 121 - Check the predictor result based on the actual class value using a string match. Then increment the correct counter and output each test's results to the console
7. Line 127-128 - Calculate and print the accuracy score

2.5 The Dataset

The dataset chosen for this implementation is the well-known iris dataset. The data set contains 150 records based on real-life iris observations made in 1936 by British statistician/biologist Ronald Fisher *(UCI Machine Learning Repository: Iris Data Set, n.d.)*.

The dataset has the following 4 attributes, with all measurements being in cm

I. sepal length
II. sepal width
III. petal length
IV. petal width

Each record corresponds with one of 3 classes, representing three types of iris species

I. Setosa
II. Versicolour
III. Virginica

The dataset is iconic due to the clear linear separation between classes that can be visualised. The separation is ideal for this implementation as it will allow the accuracy of the learner to be evaluated with simple metrics.

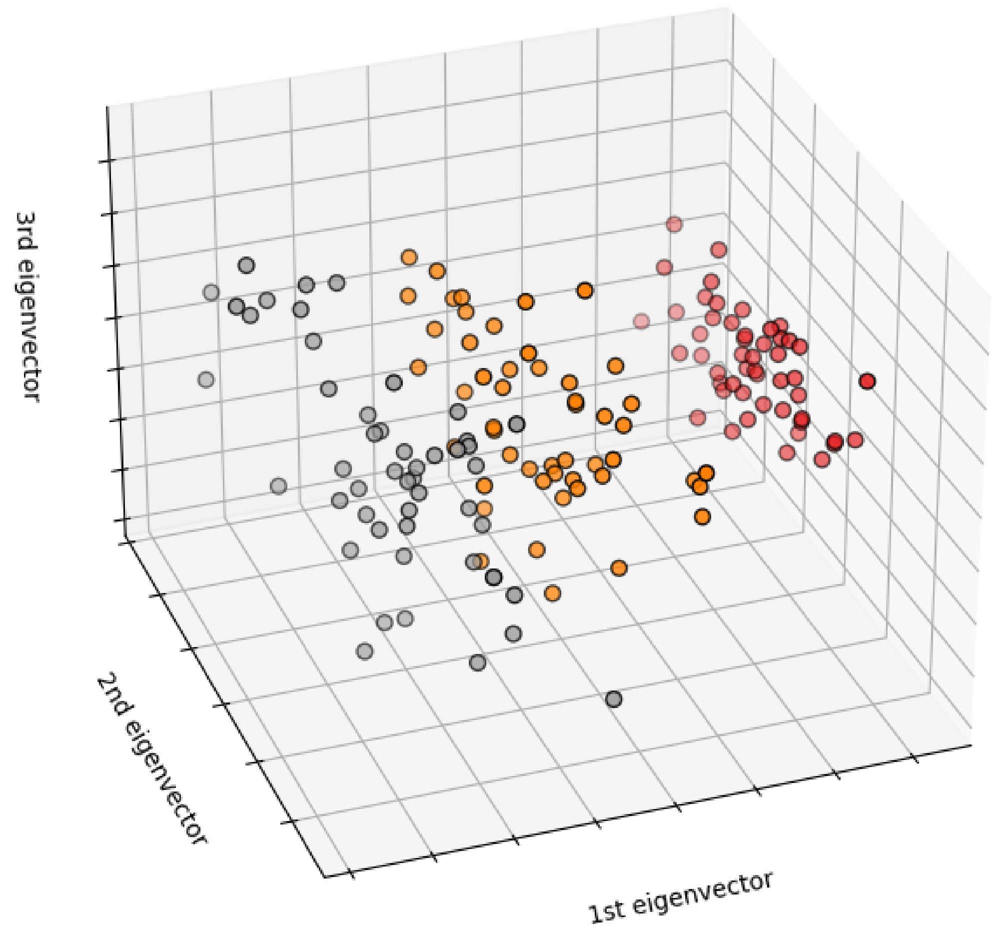


Figure #1 - Separation of classes in iris dataset, image credit (scikit-learn, 2021)

2.6 Data Preparation

To prepare the data for input into the algorithm, the data needs to be split into train and testing datasets. Due to the small size of the total dataset, it is recommended to have a higher proportion ($> 66\%$) of the data allocated to the training set (Dobbin and Simon, 2011)

For this implementation, a **70:30** train: test split has been used. There are 45 records in the testing set as $150 \ * \ 0.30 \ = \ 45$. As the dataset is rather small, data splitting was done manually by copying text into two files. The two files are *iris.dat* (training) and *test.dat* (testing).

2.7 The choice of a k value

In the determination of k, the following methods were used:

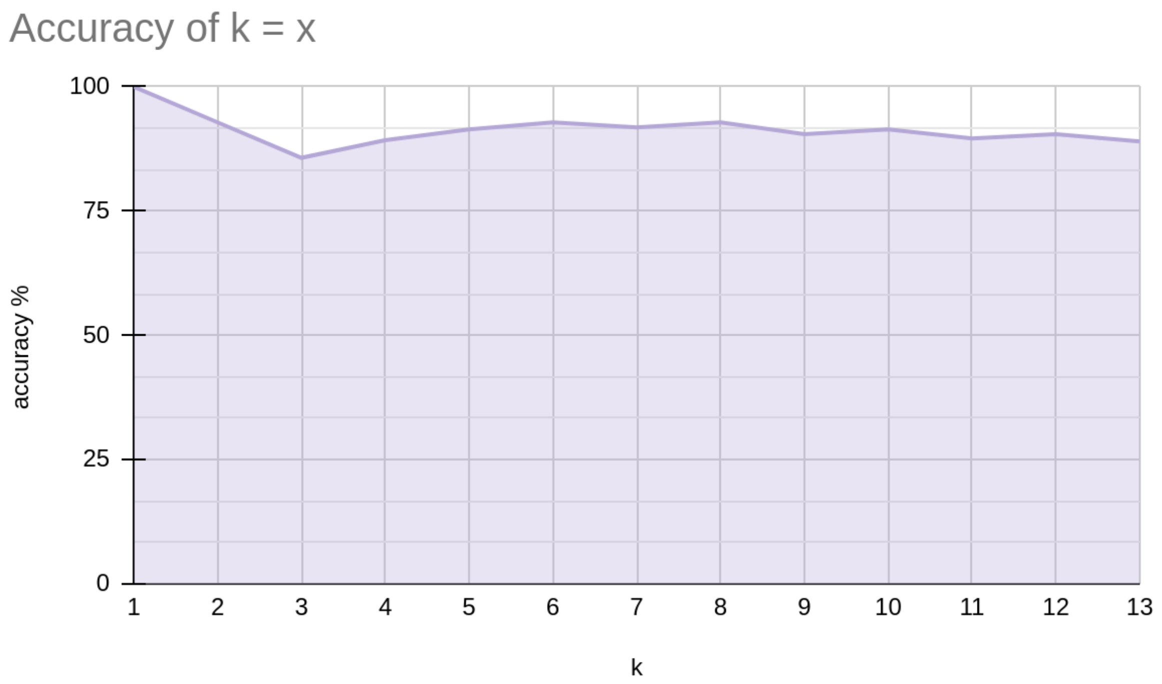
I. **Square Root Method** - a square root of all the samples in the training set. Round to the nearest odd number.

$$\sqrt{150} = 12.25$$
$$\therefore K = 11$$

II. **Cross-Validation** - Start with $k = 1$ and increment k until the accuracy stabilises. Further increasing of k will lead to a decline in accuracy, pick the k value at the start of stable accuracy measures.

The results of these experiments can be seen here:

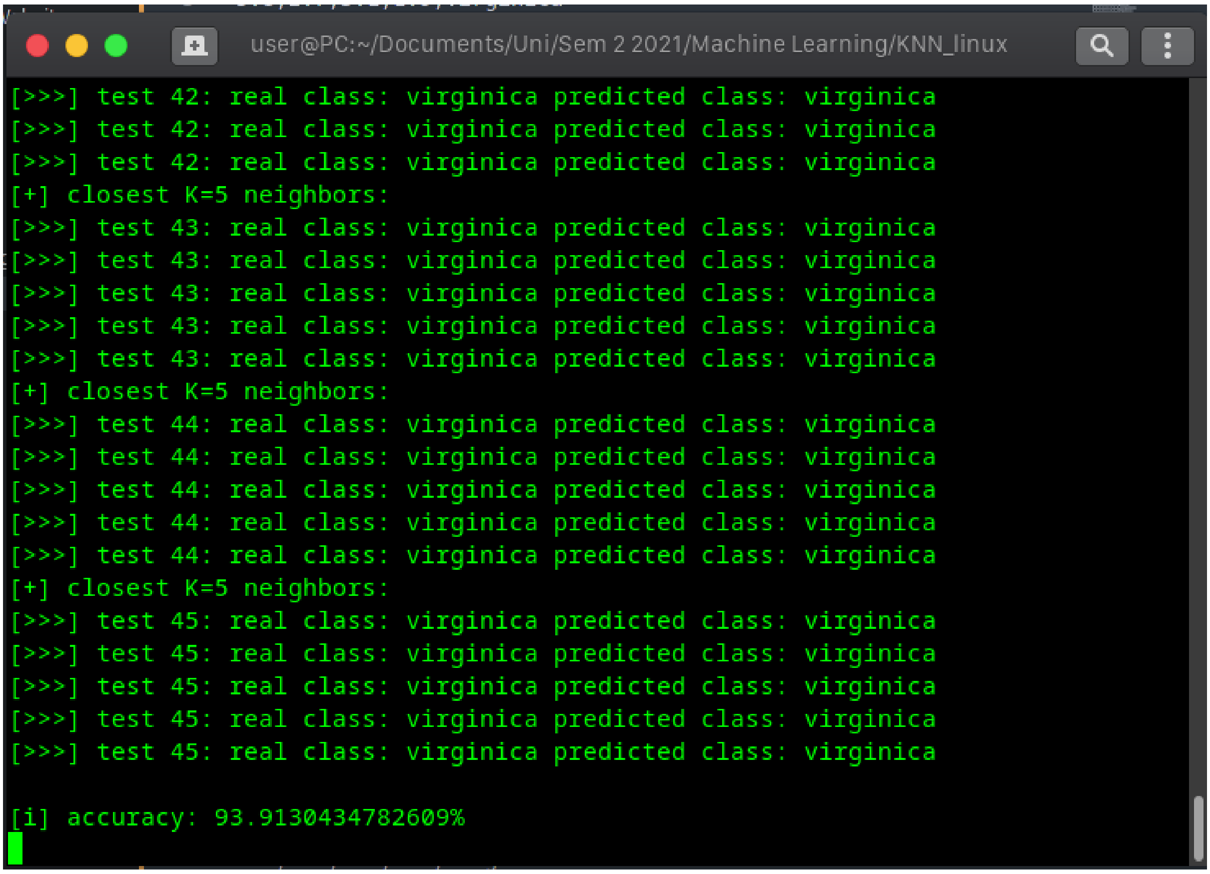
Accuracy scores for $k = x$, accuracy measured in per cent (%)												
1	2	3	4	5	6	7	8	9	10	11	12	13
100	92.85	85.71	89.26	91.43	92.86	91.83	92.86	90.48	91.43	89.61	90.47	89.01



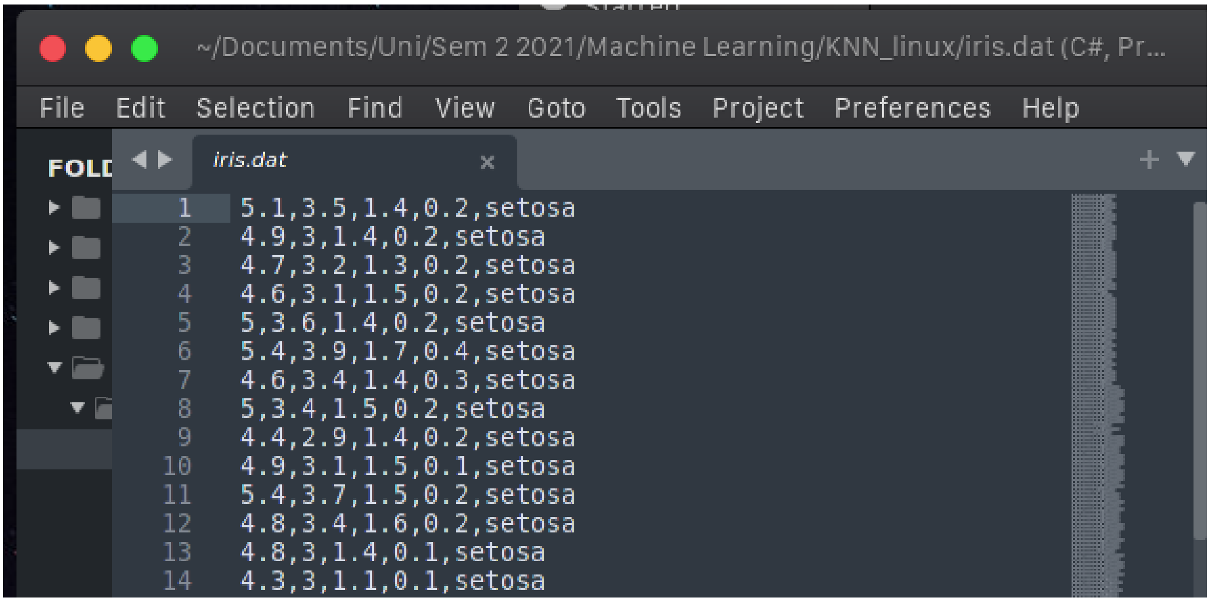
Additionally, *k* should be an odd integer to avoid ties when the neighbours vote. Using the cross-validation method the best option for k is $k = 5$. The square root method produces an even number, rounding down to $k = 11$ produces a worse accuracy than using the cross-validation result, hence that is the *k* value chosen.

3.0 Evaluation

The algorithm is working as expected. All functions implemented within the code appear to be error-free and produce the expected results. The console output is able to show the vote each k neighbour gives for each classification. Since there are 45 values in the testing set, we would expect to see 45 tests run with 5 neighbour votes each. Below is the console output for this implementation showing the working result:



Pictured below is the testing partition of the iris dataset. The file format is a .dat, based on a conversion from the .csv file that can be downloaded. The file is located in the same directory as the executables as this is the relative path specified in the code.



Testing has been done on Windows and Linux, with different executables provided due to differences in the mono compiler on Linux and the .NET used by visual studio. Once the separate executables were made, the implementation works on both platforms.

4.0 Conclusion

The k-nearest neighbour algorithm (kNN), which classifies data points according to the majority vote of its neighbours has been successfully implemented in this report using C#. k-nn is a good choice for a machine learning algorithm due to its simple principle and high accuracy.

4.1 Challenges

In the implementation of the code, there were many challenges such as issues with the stream reader, compiling and loop iterations. However, all these challenges were solved with persistence to result in a working implementation.

4.2 Effectiveness with Data

The algorithm with $k = 5$ was able to classify with an accuracy score of 93.91% based on the 70:30 split. The accuracy score achieved shows the algorithm was very effective in the tested supervised learning scenario with the iris dataset.

5. References

1. www.javatpoint.com. 2021. *K-Nearest Neighbor(KNN) Algorithm for Machine Learning*. [online] Available at: <https://www.javatpoint.com/k-nearest-neighbor-algorithm-for-machine-learning> [Accessed 20 September 2021].
2. Fix, E. and Hodges, J., 1951. Discriminatory Analysis. Nonparametric Discrimination: Consistency Properties. [online] Randolph Field: USAF School of Aviation. Available at: <https://apps.dtic.mil/dtic/tr/fulltext/u2/a800276.pdf> [Accessed 20 September 2021].
3. En.wikipedia.org. 2021. *k-nearest neighbors algorithm - Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm> [Accessed 20 September 2021].
4. Badole, M., 2021. *How KNN Uses Distance Measures? - Analytics Vidhya*. [online] Analytics Vidhya. Available at: <https://www.analyticsvidhya.com/blog/2021/08/how-knn-uses-distance-measures/> [Accessed 20 September 2021].
5. Paul, R., 2020. Euclidean Distance and Normalization of a Vector. [online] Medium. Available at: <https://paulrohan.medium.com/euclidean-distance-and-normalization-of-a-vector-76f7a97abd9> [Accessed 25 September 2021].
6. Archive.ics.uci.edu. n.d. *UCI Machine Learning Repository: Iris Data Set*. [online] Available at: <https://archive.ics.uci.edu/ml/datasets/Iris> [Accessed 26 September 2021].
7. Dobbin, K. and Simon, R., 2011. Optimally splitting cases for training and testing high dimensional classifiers. *BMC Medical Genomics*, [online] 4(1). Available at: <https://brb.nci.nih.gov/techreport/Dobbin-SampleSplitting.pdf> [Accessed 29 September 2021].