**Understanding k-NN Classification Using C#**

By [James McCaffrey](https://learn.microsoft.com/en-us/archive/msdn-magazine/2017/december/%5Carchive%5Cmsdn-magazine%5Cauthors%5CJames_McCaffrey)

The goal of a classification problem is to use the values of two or more predictor variables (often called features in machine learning [ML] terminology) to predict a class (sometimes called a label). For example, you might want to predict the risk of a server failing (low, medium, high) based on the average number of HTP requests handled and the physical temperature of the server.

There are many ML classification techniques, including naive Bayes classification, neural network classification, and decision tree classification. The k-nearest neighbors (k-NN) algorithm is a relatively simple and elegant approach. Relative to other techniques, the advantages of k-NN classification are simplicity and flexibility. The two primary disadvantages are that k-NN doesn’t work well with non-­numeric predictor values, and it doesn’t scale well to huge data sets.

This article explains exactly how k-NN classification works and presents an end-to-end demo program written in C#. The best way to see where this article is headed is to take a look at the demo program in **Figure 1**. The demo problem is to predict the class (“0,” “1,” “2”) of an item that has two predictor variables with values (5.25, 1.75). If k (the number of neighbor values to examine) is set to 1, then the predicted class is “1.” If k is set to 4, the predicted class is “2.” Behind the scenes, the demo program uses a set of 33 training data items to make the predictions.

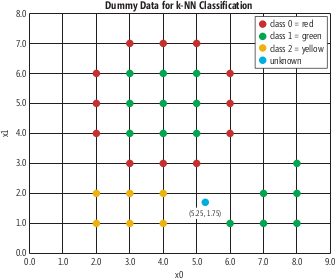
**Figure 1 Classification Demo Using the k-NN Algorithm**

Many ML libraries have built-in k-NN classification functions, but library functions can be difficult or impossible (due to legal issues) to integrate into a production system. Understanding how to implement k-NN classification from scratch gives you full control, and the ability to customize your system.

This article assumes you have intermediate or better programming skills, but doesn’t assume you know anything about k-NN classification. The demo program is coded using C#, but you shouldn’t have too much trouble refactoring the code to another language, such as Java or Python. The demo program is a bit too long to present in its entirety, but the complete source code is available in the file download that accompanies this article.

**Understanding the k-NN Algorithm**

The graph in **Figure 2** represents the data used in the demo program. There are 33 training items. Each item has two predictor values, x0 and x1. The k-NN algorithm can handle problems with any number of predictors, but the demo uses just two so that the problem can be easily visualized in a graph.

  
**Figure 2 Demo Program Training Data**

The values of the predictor variables are all between 0 and 9. When performing k-NN classification, it’s important to normalize the predictor values so that large magnitudes don’t overwhelm small magnitudes. For example, suppose you have predictor variables annual income (such as $48,000) and age (such as 32). You could normalize by dividing all income values by 10,000 (giving values like 4.8) and dividing all age values by 10 (giving values like 3.2). Two other common normalization techniques are z-score normalization and min-max normalization.

Many ML techniques, including k-NN classification, require training data that has known, correct predictor values and class labels. The demo training data has three different classes, indicated by red, green and yellow. The blue data point at (5.25, 1.75) is the unknown to predict. When k is set to 1, k-NN finds the single, closest training data item and returns the class of that closest data item as the predicted class of the unknown item.

In this example, for the (5.25, 1.75) unknown item, the closest training data item is the green (class = “1”) dot at (6.0, 1.0) so the predicted class is “1.” When using k-NN, you must specify how to measure distance between data items so you can define what “closest” means. The demo program uses Euclidean distance. The distance between (5.25, 1.75) and (6.0, 1.0) is sqrt((5.25 - 6.0)^2 + (1.75 - 1.0)^2 ) = sqrt(0.5625 + 0.5625) = sqrt(1.1250) = 1.061, as displayed in **Figure 1**.

When k is set to 4, the predicted class depends on the four nearest training data items. In the demo example, the four closest items are at (6.0, 1.0), (5.0, 3.0), (4.0, 2.0) and (4.0, 1.0). The associated class labels for the items are (“1,” “0,” “2,” “2”). Because there are more “2” labels than “0” and “1” labels, the predicted class is “2.”

When using k-NN you must specify how to determine the predicted class from the set of the k closest class labels. The demo program uses a majority-vote approach. Notice that when using a simple majority vote, you can run into tie situations. In practice, however, k-NN majority-vote ties are relatively rare. The demo program returns the lowest class label in case of ties. For example, if the associated labels are (“2,” “1,” “1,” “2”), then the demo program voting mechanism would return “1.”

**The Demo Program**

To code the demo program, I launched Visual Studio and created a new C# console application program and named it KNN. I used Visual Studio 2015, but the demo program has no significant .NET Framework dependencies so any recent version of Visual Studio will work fine.

After the template code loaded into the editor window, I right-clicked on file Program.cs in the Solution Explorer window and renamed the file to KNNProgram.cs, then allowed Visual Studio to automatically rename class Program for me. At the top of the template-generated code, I deleted all unnecessary using statements, leaving just the one that references the top-level System namespace.

The overall program structure, with a few minor edits to save space, is presented in **Figure 3**. The demo uses a simple static method approach, and all normal error checking has been removed to keep the main ideas clear.

Figure 3 Overall Program Structure

using System;

namespace KNN

{

  class KNNProgram

  {

    static void Main(string[] args)

    {

      Console.WriteLine("Begin k-NN classification demo ");

      double[][] trainData = LoadData();

      int numFeatures = 2;

      int numClasses = 3;

      double[] unknown = new double[] { 5.25, 1.75 };

      Console.WriteLine("Predictor values: 5.25 1.75 ");

      int k = 1;

      Console.WriteLine("With k = 1");

      int predicted = Classify(unknown, trainData,

        numClasses, k);

      Console.WriteLine("Predicted class = " + predicted);

      k = 4;

      Console.WriteLine("With k = 4");

      predicted = Classify(unknown, trainData,

        numClasses, k);

      Console.WriteLine("Predicted class = " + predicted);

      Console.WriteLine("End k-NN demo ");

      Console.ReadLine();

    }

    static int Classify(double[] unknown,

      double[][] trainData, int numClasses, int k) { . . }

    static int Vote(IndexAndDistance[] info,

      double[][] trainData, int numClasses, int k) { . . }

    static double Distance(double[] unknown,

      double[] data) { . . }

    static double[][] LoadData() { . . }

  } // Program class

  public class IndexAndDistance : IComparable<IndexAndDistance>

  {

    public int idx;  // Index of a training item

    public double dist;  // To unknown

    // Need to sort these to find k closest

    public int CompareTo(IndexAndDistance other)

    {

      if (this.dist < other.dist) return -1;

      else if (this.dist > other.dist) return +1;

      else return 0;

    }

  }

} // ns

The demo begins by setting up the training data:

C#Copy

static void Main(string[] args)

{

  Console.WriteLine(“Begin k-NN classification demo “);

  double[][] trainData = LoadData();

...

The data is hardcoded and stored into an array-of-arrays-style matrix. In a non-demo scenario you’d likely read data into memory from a text file. Because k-NN classification typically stores all training data into memory, the technique doesn’t easily scale to scenarios with very large data sets. Method LoadData is defined as:

C#Copy

static double[][] LoadData()

{

  double[][] data = new double[33][];

  data[0] = new double[] { 2.0, 4.0, 0 };

    ...

  data[12] = new double[] { 3.0, 4.0, 1 };

    ...

  data[32] = new double[] { 4.0, 2.0, 2 };

  return data;

}

The first two values of each item are the predictor values and the last value is the class label. A common alternative design is to store predictor values in a matrix, but store the class labels in a separate array. The demo code assumes the class labels are numeric, and consecutively numbered starting at 0. If your class labels are non-numeric, such as “low,” “medium,” “high,” you must encode the data, either in a preprocessing stage, or programmatically when reading the data into memory.

Next, the demo sets up the unknown item to predict, like so:

C#Copy

int numFeatures = 2;

int numClasses = 3;

double[] unknown = new double[] { 5.25, 1.75 };

Console.WriteLine("Predictor values: 5.25 1.75 ");

The demo program doesn’t actually use the numFeatures variable, but as you’ll see shortly, there are many possible customization points for k-NN classification, and a numFeatures value can be useful.

Next, the demo makes the simplest possible k-NN prediction:

C#Copy

int k = 1;

Console.WriteLine("With k = 1");

int predicted = Classify(unknown, trainData,

  numClasses, k);

Console.WriteLine("Predicted class = " + predicted);

The Classify method accepts parameters for the item to predict, a matrix of training data, the number of classes in the training data, and the number of nearest neighbors to evaluate. In principle, you could implement method Classify so that it scans the training data to programmatically determine the number of classes, but the effort required outweighs the benefit, in my opinion.

The demo program concludes with:

C#Copy

...

  k = 4;

  Console.WriteLine(“With k = 4”);

  predicted = Classify(unknown, trainData,

    numClasses, k);

  Console.WriteLine(“Predicted class = “ + predicted);

  Console.WriteLine(“End k-NN demo “);

  Console.ReadLine();

}

There aren’t any good rules of thumb for determining the value of k when using k-NN classification. The value of k in k-NN classification is a hyperparameter and must be determined by intuition and experimentation.

**Implementing the k-NN Algorithm**

In very high-level pseudo-code, the k-NN classification algorithm used by the demo is:

XMLCopy

Compute distances from unknown to all training items

Sort the distances from nearest to farthest

Scan the k-nearest items; use a vote to determine the result

The definition of method Classify begins by computing and storing into an array the distances between the unknown item to classify and all training items, as shown in **Figure 4**.

Figure 4 Definition of the Classify Method

C#Copy

static int Classify(double[] unknown,

  double[][] trainData, int numClasses, int k)

{

  int n = trainData.Length;

  IndexAndDistance[] info = new IndexAndDistance[n];

  for (int i = 0; i < n; ++i) {

    IndexAndDistance curr = new IndexAndDistance();

    double dist = Distance(unknown, trainData[i]);

    curr.idx = i;

    curr.dist = dist;

    info[i] = curr;

  }

...

It’s not enough to store and sort just the distances from the unknown item to each training item, because you need to know the class associated with each distance. The demo program defines a simple container class, named IndexAndDistance, that holds an index to a training item and the associated distance to the unknown item:

C#Copy

public class IndexAndDistance : IComparable<IndexAndDistance>

{

  public int idx;  // index of a training item

  public double dist;  // distance to unknown

  public int CompareTo(IndexAndDistance other) {

    if (this.dist < other.dist) return -1;

    else if (this.dist > other.dist) return +1;

    else return 0;

  }

}

The class label is stored indirectly because if you know the index of a training item, you can look up the class label in the matrix of training data as the last cell in the row pointed to by the index. An alternative design is to store just the class label explicitly, but storing the training item index gives you more flexibility. Another alternative is to explicitly store both the index and the class label.

Because k-NN needs to sort the index-distance items to deter­mine the k-nearest items, the IndexAndDistance definition implements the IComparable interface by defining a CompareTo method. Doing this allows you to automatically sort an array of IndexAndDistance objects. If you refactor the demo code to a programming language that doesn’t support auto-sorting, you’ll have to implement a custom sort method that operates on a structure, or implement a custom sort method that works with parallel arrays.

There are a few alternatives to sorting index-distance items—for example, using a heap data structure—but in my opinion the increased complexity outweighs any performance improvement you’d get.

After all training index-distance items are stored, they’re sorted, and information for the k-nearest items is displayed:

C#Copy

Array.Sort(info);  // sort by distance

Console.WriteLine("Nearest / Distance / Class");

Console.WriteLine("==========================");

for (int i = 0; i < k; ++i) {

  int c = (int)trainData[info[i].idx][2];

  string dist = info[i].dist.ToString("F3");

  Console.WriteLine("( " + trainData[info[i].idx][0] +

    "," + trainData[info[i].idx][1] + " )  :  " +

    dist + "        " + c);

}

The class label, c, is extracted from cell [2] of a row of training data. This assumes there are two predictor variables. A superior approach would be to pass a numFeatures argument to method Classify, then access cell [numFeatures].

Displaying information about the k-nearest training items is optional, but it illustrates an advantage of k-NN classification compared to many other techniques. The k-NN algorithm is somewhat interpretable in the sense that you can determine exactly how an unknown item was classified. Some techniques, notably neural network classification, are difficult or impossible to interpret.

The Classify method concludes by scanning the k-nearest training items to determine a predicted class for the unknown item:

C#Copy

...

  int result = Vote(info, trainData, numClasses, k);

  return result;

} // Classify

Helper method Vote accepts the array of all index-distance items. An alternative approach is to pass just the first k cells of the array.

**Distance and Voting**

The Classify method calls helper methods Distance and Vote. Method Distance is defined as:

C#Copy

static double Distance(double[] unknown, double[] data)

{

  double sum = 0.0;

  for (int i = 0; i < unknown.Length; ++i)

    sum += (unknown[i] - data[i]) \* (unknown[i] - data[i]);

  return Math.Sqrt(sum);

}

This is a simple implementation of Euclidean distance. Common alternatives you can consider include Manhattan distance, Mahalanobis distance and measures based on similarity, such as the radial basis function kernel. Because k-NN needs a notion of “nearest” and most distance metrics work with strictly numeric or strictly non-numeric data, k-NN classification is not well-suited for problems with mixed numeric and categorical data.

Helper method Vote is presented in **Figure 5**. Determining a consensus class label from k items is a bit trickier than you might guess. The demo uses the simplest approach, where each of the k-nearest training items gets one vote for its class label. This approach doesn’t take into account the distance. A common alternative is to weight votes by distance so that training items that are closer to the unknown item have more influence.

Figure 5 The Vote Method

C#Copy

static int Vote(IndexAndDistance[] info, double[][] trainData,

  int numClasses, int k)

{

  int[] votes = new int[numClasses];  // One cell per class

  for (int i = 0; i < k; ++i) {       // Just first k

    int idx = info[i].idx;            // Which train item

    int c = (int)trainData[idx][2];   // Class in last cell

    ++votes[c];

  }

  int mostVotes = 0;

  int classWithMostVotes = 0;

  for (int j = 0; j < numClasses; ++j) {

    if (votes[j] > mostVotes) {

      mostVotes = votes[j];

      classWithMostVotes = j;

    }

  }

  return classWithMostVotes;

}

The demo implementation doesn’t explicitly deal with duplicate training-data items. Much real-life data doesn’t have an excessive amount of duplicate data, but if your training data does include lots of duplicates, you might want to consider removing them.

**Wrapping Up**

The k-NN classification algorithm is arguably one of the simplest of all machine learning techniques. In addition to simplicity, k-NN classification can easily deal with multi-class problems (as opposed to some techniques that work easily only for binary classification). Another advantage is that k-NN can easily deal with data that has unusual characteristics, such as the dummy demo data that has several pockets of class labels. Some techniques, such as logistic regression and non-kernel support vector machines, can deal only with data that is linearly separable. Two other relative advan­tages of k-NN classification are flexibility of implementation, and interpretability of results.

On the negative side, k-NN classification doesn’t work well with categorical data or mixed numeric and categorical data. The technique doesn’t scale well to huge data sets. And k-NN classification is highly sensitive to the local geometry of the training data.

When I have a classification problem with strictly numeric predictor values, and a non-huge set of training data (for example, less than one million items), using k-NN is often my first approach. Then I’ll try more sophisticated techniques, typically including neural network classification. In some situations, an ensemble approach that combines the results of k-NN classification with neural network classification can lead to a very robust and accurate prediction system.