### k-Nearest Neighbors

A simple but powerful approach for making predictions is to use the most similar historical examples to the new data. This is the principle behind the k-Nearest Neighbors algorithm.

After completing this tutorial you will know:

* How to code the k-Nearest Neighbors algorithm step-by-step.
* How to evaluate k-Nearest Neighbors on a real dataset.
* How to use k-Nearest Neighbors to make a prediction for new data.

The k-Nearest Neighbors algorithm or KNN for short is a very simple technique.

The entire training dataset is stored. When a prediction is required, the k-most similar records to a new record from the training dataset are then located. From these neighbors, a summarized prediction is made.

Similarity between records can be measured many different ways. A problem or data-specific method can be used. Generally, with tabular data, a good starting point is the [Euclidean distance](https://en.wikipedia.org/wiki/Euclidean_distance).

Once the neighbors are discovered, the summary prediction can be made by returning the most common outcome or taking the average. As such, KNN can be used for classification or regression problems.

There is no model to speak of other than holding the entire training dataset. Because no work is done until a prediction is required, KNN is often referred to as a lazy learning method.

### Iris Flower Species Dataset

In this tutorial we will use the Iris Flower Species Dataset.

The Iris Flower Dataset involves predicting the flower species given measurements of iris flowers.

It is a multiclass classification problem. The number of observations for each class is balanced. There are 150 observations with 4 input variables and 1 output variable. The variable names are as follows:

* Sepal length in cm.
* Sepal width in cm.
* Petal length in cm.
* Petal width in cm.
* Class

A sample of the first 5 rows is listed below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | 5.1,3.5,1.4,0.2,Iris-setosa  4.9,3.0,1.4,0.2,Iris-setosa  4.7,3.2,1.3,0.2,Iris-setosa  4.6,3.1,1.5,0.2,Iris-setosa  5.0,3.6,1.4,0.2,Iris-setosa  ... |

The baseline performance on the problem is approximately 33%.

Download the dataset and save it into your current working directory with the filename “iris.csv“.

* [Download Dataset (iris.csv)](https://raw.githubusercontent.com/jbrownlee/Datasets/master/iris.csv)

## k-Nearest Neighbors (in 3 easy steps)

First we will develop each piece of the algorithm in this section, then we will tie all of the elements together into a working implementation applied to a real dataset in the next section.

This k-Nearest Neighbors tutorial is broken down into 3 parts:

* **Step 1**: Calculate Euclidean Distance.
* **Step 2**: Get Nearest Neighbors.
* **Step 3**: Make Predictions.

These steps will teach you the fundamentals of implementing and applying the k-Nearest Neighbors algorithm for classification and regression predictive modeling problems.

### Step 1: Calculate Euclidean Distance

The first step is to calculate the distance between two rows in a dataset.

Rows of data are mostly made up of numbers and an easy way to calculate the distance between two rows or vectors of numbers is to draw a straight line. This makes sense in 2D or 3D and scales nicely to higher dimensions.

We can calculate the straight line distance between two vectors using the Euclidean distance measure. It is calculated as the square root of the sum of the squared differences between the [two vectors](https://machinelearningmastery.com/gentle-introduction-vectors-machine-learning/).

* Euclidean Distance = sqrt(sum i to N (x1\_i – x2\_i)^2)

Where x1 is the first row of data, x2 is the second row of data and i is the index to a specific column as we sum across all columns.

With Euclidean distance, the smaller the value, the more similar two records will be. A value of 0 means that there is no difference between two records.

Below is a function named euclidean\_distance() that implements this in Python.



|  |  |
| --- | --- |
| 1  2  3  4  5  6 | # calculate the Euclidean distance between two vectors  def euclidean\_distance(row1, row2):  distance = 0.0  for i in range(len(row1)-1):  distance += (row1[i] - row2[i])\*\*2  return sqrt(distance) |

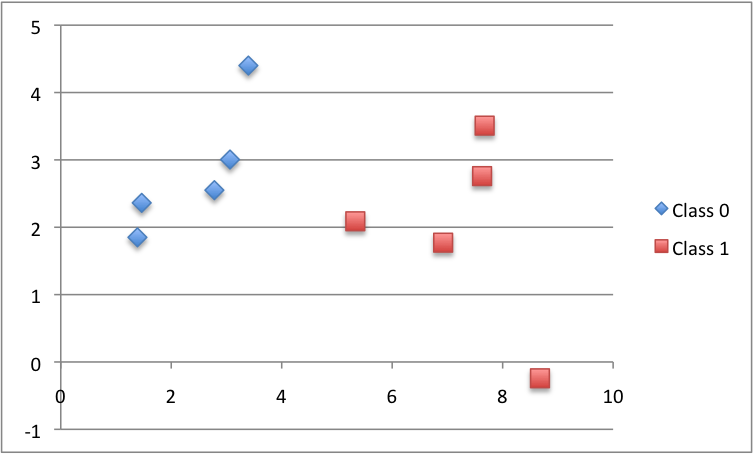
You can see that the function assumes that the last column in each row is an output value which is ignored from the distance calculation.

We can test this distance function with a small contrived classification dataset. We will use this dataset a few times as we construct the elements needed for the KNN algorithm.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | X1 X2 Y  2.7810836 2.550537003 0  1.465489372 2.362125076 0  3.396561688 4.400293529 0  1.38807019 1.850220317 0  3.06407232 3.005305973 0  7.627531214 2.759262235 1  5.332441248 2.088626775 1  6.922596716 1.77106367 1  8.675418651 -0.242068655 1  7.673756466 3.508563011 1 |

Below is a plot of the dataset using different colors to show the different classes for each point.



Scatter Plot of the Small Contrived Dataset for Testing the KNN Algorithm

Putting this all together, we can write a small example to test our distance function by printing the distance between the first row and all other rows. We would expect the distance between the first row and itself to be 0, a good thing to look out for.

The full example is listed below.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | # Example of calculating Euclidean distance  from math import sqrt    # calculate the Euclidean distance between two vectors  def euclidean\_distance(row1, row2):  distance = 0.0  for i in range(len(row1)-1):  distance += (row1[i] - row2[i])\*\*2  return sqrt(distance)    # Test distance function  dataset = [[2.7810836,2.550537003,0],  [1.465489372,2.362125076,0],  [3.396561688,4.400293529,0],  [1.38807019,1.850220317,0],  [3.06407232,3.005305973,0],  [7.627531214,2.759262235,1],  [5.332441248,2.088626775,1],  [6.922596716,1.77106367,1],  [8.675418651,-0.242068655,1],  [7.673756466,3.508563011,1]]  row0 = dataset[0]  for row in dataset:  distance = euclidean\_distance(row0, row)  print(distance) |

Running this example prints the distances between the first row and every row in the dataset, including itself.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | 0.0  1.3290173915275787  1.9494646655653247  1.5591439385540549  0.5356280721938492  4.850940186986411  2.592833759950511  4.214227042632867  6.522409988228337  4.985585382449795 |

Now it is time to use the distance calculation to locate neighbors within a dataset.

### Step 2: Get Nearest Neighbors

Neighbors for a new piece of data in the dataset are the k closest instances, as defined by our distance measure.

To locate the neighbors for a new piece of data within a dataset we must first calculate the distance between each record in the dataset to the new piece of data. We can do this using our distance function prepared above.

Once distances are calculated, we must sort all of the records in the training dataset by their distance to the new data. We can then select the top k to return as the most similar neighbors.

We can do this by keeping track of the distance for each record in the dataset as a tuple, sort the list of tuples by the distance (in descending order) and then retrieve the neighbors.

Below is a function named get\_neighbors() that implements this.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | # Locate the most similar neighbors  def get\_neighbors(train, test\_row, num\_neighbors):  distances = list()  for train\_row in train:  dist = euclidean\_distance(test\_row, train\_row)  distances.append((train\_row, dist))  distances.sort(key=lambda tup: tup[1])  neighbors = list()  for i in range(num\_neighbors):  neighbors.append(distances[i][0])  return neighbors |

You can see that the euclidean\_distance() function developed in the previous step is used to calculate the distance between each train\_row and the new test\_row.

The list of train\_row and distance tuples is sorted where a custom key is used ensuring that the second item in the tuple (tup[1]) is used in the sorting operation.

Finally, a list of the num\_neighbors most similar neighbors to test\_row is returned.

We can test this function with the small contrived dataset prepared in the previous section.

The complete example is listed below.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | # Example of getting neighbors for an instance  from math import sqrt    # calculate the Euclidean distance between two vectors  def euclidean\_distance(row1, row2):  distance = 0.0  for i in range(len(row1)-1):  distance += (row1[i] - row2[i])\*\*2  return sqrt(distance)    # Locate the most similar neighbors  def get\_neighbors(train, test\_row, num\_neighbors):  distances = list()  for train\_row in train:  dist = euclidean\_distance(test\_row, train\_row)  distances.append((train\_row, dist))  distances.sort(key=lambda tup: tup[1])  neighbors = list()  for i in range(num\_neighbors):  neighbors.append(distances[i][0])  return neighbors    # Test distance function  dataset = [[2.7810836,2.550537003,0],  [1.465489372,2.362125076,0],  [3.396561688,4.400293529,0],  [1.38807019,1.850220317,0],  [3.06407232,3.005305973,0],  [7.627531214,2.759262235,1],  [5.332441248,2.088626775,1],  [6.922596716,1.77106367,1],  [8.675418651,-0.242068655,1],  [7.673756466,3.508563011,1]]  neighbors = get\_neighbors(dataset, dataset[0], 3)  for neighbor in neighbors:  print(neighbor) |

Running this example prints the 3 most similar records in the dataset to the first record, in order of similarity.

As expected, the first record is the most similar to itself and is at the top of the list.



|  |  |
| --- | --- |
| 1  2  3 | [2.7810836, 2.550537003, 0]  [3.06407232, 3.005305973, 0]  [1.465489372, 2.362125076, 0] |

Now that we know how to get neighbors from the dataset, we can use them to make predictions.

### Step 3: Make Predictions

The most similar neighbors collected from the training dataset can be used to make predictions.

In the case of classification, we can return the most represented class among the neighbors.

We can achieve this by performing the max() function on the list of output values from the neighbors. Given a list of class values observed in the neighbors, the max() function takes a set of unique class values and calls the count on the list of class values for each class value in the set.

Below is the function named predict\_classification() that implements this.



|  |  |
| --- | --- |
| 1  2  3  4  5  6 | # Make a classification prediction with neighbors  def predict\_classification(train, test\_row, num\_neighbors):  neighbors = get\_neighbors(train, test\_row, num\_neighbors)  output\_values = [row[-1] for row in neighbors]  prediction = max(set(output\_values), key=output\_values.count)  return prediction |

We can test this function on the above contrived dataset.

Below is a complete example.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | # Example of making predictions  from math import sqrt    # calculate the Euclidean distance between two vectors  def euclidean\_distance(row1, row2):  distance = 0.0  for i in range(len(row1)-1):  distance += (row1[i] - row2[i])\*\*2  return sqrt(distance)    # Locate the most similar neighbors  def get\_neighbors(train, test\_row, num\_neighbors):  distances = list()  for train\_row in train:  dist = euclidean\_distance(test\_row, train\_row)  distances.append((train\_row, dist))  distances.sort(key=lambda tup: tup[1])  neighbors = list()  for i in range(num\_neighbors):  neighbors.append(distances[i][0])  return neighbors    # Make a classification prediction with neighbors  def predict\_classification(train, test\_row, num\_neighbors):  neighbors = get\_neighbors(train, test\_row, num\_neighbors)  output\_values = [row[-1] for row in neighbors]  prediction = max(set(output\_values), key=output\_values.count)  return prediction    # Test distance function  dataset = [[2.7810836,2.550537003,0],  [1.465489372,2.362125076,0],  [3.396561688,4.400293529,0],  [1.38807019,1.850220317,0],  [3.06407232,3.005305973,0],  [7.627531214,2.759262235,1],  [5.332441248,2.088626775,1],  [6.922596716,1.77106367,1],  [8.675418651,-0.242068655,1],  [7.673756466,3.508563011,1]]  prediction = predict\_classification(dataset, dataset[0], 3)  print('Expected %d, Got %d.' % (dataset[0][-1], prediction)) |

Running this example prints the expected classification of 0 and the actual classification predicted from the 3 most similar neighbors in the dataset.



|  |  |
| --- | --- |
| 1 | Expected 0, Got 0. |

We can imagine how the predict\_classification() function can be changed to calculate the mean value of the outcome values.

Reference:

1. <https://machinelearningmastery.com/tutorial-to-implement-k-nearest-neighbors-in-python-from-scratch/>