Falling Detection

Machine Learning Using Python and kNN

Brais Galvan Sotelo (19563)
Northwestern Polytechnic University

Table of Contents

Introduction

We want to analyze the data from two sensors from mobile devices, the gyroscope and the accelerometer in order to predict if the device is falling or not.

Both sensor have x, y, z data, where the accelerometer focuses on the linear acceleration of the device and the gyroscope in the angular velocity.

We want to implement kNN in order to predict whereas the device is falling or not.

Because kNN is a supervised learning algorithm, meaning that it will learn from existing labeled data, we provide a dataset with some data, and then attempt to make a prediction in a new instance of data.

KNN Manual Implementation - Data and K value

First we have to get and understand the data.

We have the xyz coordinates of both the accelerometer and the gyroscope coupled together and the resulting outcome indicated as a “+” or “-” depending on whereas the divide falls or not.

Next we want to find the K value, which is usually calculated as the odd value closest to the square root of the number of data instances (n). In this case n = 8.

Hence $k = \sqrt{8} = 2.83 \rightarrow 3$ is the closes odd number.

Accelerometer Data			Gyroscope Data			Fall (+), Not Fall (-)
x	y	z	x	y	z	+/-
1	2	3	2	1	3	-
2	1	3	3	1	2	-
1	1	2	3	2	2	-
2	2	3	3	2	1	-
6	5	7	5	6	7	+
5	6	6	6	5	7	+
5	6	7	5	7	6	+
7	6	7	6	5	6	+

KNN Manual Implementation - Distances to neighbours (Overview)

We want to predict the outcome for the following data

Accelerometer (x = 7, y = 6, z = 5)

Gyroscope (x = 5, y = 6, z = 7)

To predict the outcome (?) of the target [7, 6, 5, 5, 6, 7, ?) we first calculate the distance to the other neighbours (or data points) for which we already know the outcome (training data). Then we will pick the k closest neighbour to make a prediction.

The distance to each neighbor is calculated by the following formula =

(Targetx1-TrainingDatax1)^2+(Targety1-TrainingDatay1)^2+
(Targetz1-TrainingDataz1)^2+(Targetx2-TrainingDatax2)^2+
(Targety2-TrainingDatay2)^2+(Targetz2-TrainingDataz2)^2

KNN Manual Implementation - Distances to neighbours (Calculatation)

Accelerometer Data			Gyroscope Data			Fall (+), Not Fall (-)	**Distance to each Neighbour** Target: (7, 6, 5, 5, 6, 7, ?)
x	y	z	x	y	z	+/-	**(7-x1)^2 + (6-y1)^2 + (5-z1)^2 + (5-x2)^2 + (6-y2) + (7-z2)^2**
1	2	3	2	1	3	-	(7-1)^2+(6-2)^2+(5-3)^2+(5-2)^2+(6-1)^2+(7-3)^2 = **106**
2	1	3	3	1	2	-	(7-2)^2+(6-1)^2+(5-3)^2+(5-3)^2+(6-1)^2+(7-2)^2 = **108**
1	1	2	3	2	2	-	(7-1)^2+(6-1)^2+(5-2)^2+(5-3)^2+(6-2)^2+(7-2)^2 = **115**
2	2	3	3	2	1	-	(7-2)^2+(6-2)^2+(5-3)^2+(5-3)^2+(6-2)^2+(7-1)^2 = **101**
6	5	7	5	6	7	+	(7-6)^2+(6-5)^2+(5-7)^2+(5-5)^2+(6-6)^2+(7-7)^2 = **6**
5	6	6	6	5	7	+	(7-5)^2+(6-6)^2+(5-6)^2+(5-6)^2+(6-5)^2+(7-7)^2 = **7**
5	6	7	5	7	6	+	(7-5)^2+(6-6)^2+(5-7)^2+(5-5)^2+(6-7)^2+(7-6)^2 = **10**
7	6	7	6	5	6	+	(7-7)^2+(6-6)^2+(5-7)^2+(5-6)^2+(6-5)^2+(7-6)^2 = **7**

KNN Manual Implementation - k closest neighbours and prediction.

We picked the k(3) closest neighbours based on the distance calculations. The closest neighbours are those with the smaller distance to the target.

Then we look at the Fall(+) Not Fall(-) data for the 3 neighbours, and predict that the outcome (?) of the target will be the most common outcome from the closest neighbours. In this case, all the neighbours are “+” (Fall), meaning that we predict the target to be “+” as well.

Accelerometer Data			Gyroscope Data			Fall (+), Not Fall (-)	**Distance to each Neighbour** Target: (7, 6, 5, 5, 6, 7, ?)
x	y	z	x	y	z	+/-	
6	5	7	5	6	7	+	**6**
5	6	6	6	5	7	+	**7**
7	6	7	6	5	6	+	**7**
7	6	5	5	6	7	**? = +**	

KNN Python Implementation - Defining Methods (Calculate Distances)

```
# Example of making predictions
from math import sqrt
```

```
# calculate the Euclidean distance between two vectors
#     Euclidean Distance = sqrt(sum i to N (x1_i - x2_i)^2)
def euclidean_distance(row1, row2):
  distance = 0.0
  for i in range(len(row1)-1):
    distance += (row1[i] - row2[i])**2
  return sqrt(distance)
```

First we import sqrt from math, to be able use it for calculating the Euclidean Distances. Then we define the method to find the Euclidean Distance from a target row to the data row, using a for loop to iterate through all the parameters in the row while applying the formula and finally finding the square root of our total calculations.

KNN Python Implementation - Defining Methods (Find closest Neighbours)

```
# 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
```

To find the closest neighbours we are going to call the method to calculate the euclidean distance for each of the training data rows.

We add the calculated distance values to a list and then select the k closest neighbours.

KNN Python Implementation - Defining Methods (Making a Classification/Prediction)

```
  # Make a classification prediction with neighbors
# - test_row is row [7, 6, 5, 5, 6, 7]
# - num_neighbors is 3
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
```

To make the prediction, we first get the closest neighbours by calling the previous methods. Then we select the output (Fall/Not Fall) data from these neighbours, and find which of the outputs is more common based on a count. The most common output among the neighbours is set to be the prediction value.

KNN Python Implementation - Data input and Testing

```
# Test distance function
# Fall (+) is represented as 0, and Not Fall (-) as 1
dataset = [[1, 2, 3, 2, 1, 3, 1],
  [2, 1, 3, 3, 1, 2, 1],
  [1, 1, 2, 3, 2, 2, 1],
  [2, 2, 3, 3, 2, 1, 1],
  [6, 5, 7, 5, 6, 7, 0],
  [5, 6, 6, 6, 5, 7, 0],
  [5, 6, 7, 5, 7, 6, 0],
  [7, 6, 7, 6, 5, 6, 0]]

target =  [7, 6, 5, 5, 6, 7]
prediction = predict_classification(dataset, target, 3)
#    Expected 0, Got 0.
print('Expected %d, Got %d.' % ([7, 6, 5, 5, 6, 7, 0][-1], prediction))
```

```
Expected 0, Got 0.
```

Here we are simply going to input our dataset and then use the classification method to get a prediction value for our target.

In this case, we predicted the value to be 0, or Fall(+), which is what we expected.

Conclusion

Comparing kNN using a manual implementation vs Python

Based on the result we concluded that both method gave the same result, namely is was predicted that the outcome for the target data is Fall(+)

The main differences between the two methods are the fact that manual calculations can be more tedious and error-prone and also scales poorly. Meaning that the larger the data set, the harder and more unreliable the manual method becomes. On the other hand, the python method scales really well, since we mostly just need to change the dataset in order to get new calculations, and we let the software do all the calculations.