Hack the North Part 3 - K Nearest Neighbors

Rushi Shah

20 September 2015

After we had the capability to compare features of curves we get from Myo sensors, our hack centered around classifying those features based on past classifications. Basically, after feeding our program some training data on what gestures you are performing and what features those gestures correspond to, the program should be able to take an arbitrary feature and predict which gesture it is.

So, K Nearest Neighbors (KNN) is a classification algorithm, essentially. Given a plot of data points, KNN will plot the unknown point and find `K` neighbors. Based on the classification of the `K` neighbors, you can hypothetically predict the classification of the unknown value.

1 Sci-Kit Learn and Nearest Neighbors Classification

Ever heard that there's a python library for everything? Well there is a python library for K Nearest Neighbors called Sci-Kit Learn (or SKLearn). Of course it does a plethora of other things as well, but we focused on the KNN capabilities.

There are a few options to keep in mind with KNN, which are well documented on the SKLearn site.

- First of all, you need to decide whether you really want a KNN. There is also a `RadiusNeighborsClassifier` that clusters data points together. This rectifies a potential bias in KNN if the number of data points for each classification are not equal. However, in our case, we have an equal number of data points in each classification, so we went with the KNN.

- Second of all, how many neighbors do you want to find? In other words, what will your `K` value be? More neighbors, and thus a higher `K`, will remove noise within data, but will also give less specific answers. In our case we decided to stick with only the closest neighbor because it represents the best guess of the classification.

- Third of all, will the k-neighbors be weighted at all? The default is uniform weights for each data point, but you can also weight based on distance or even provide your own metric. Altough it was not significant to us because we were only looking for 1 neighbor, we would set the weight based on distance.

2 Our Code

All this analysis boiled down to a very small amount of code. The return value of KNN was an index in the original array. We had a parallel array that represented the corresponding action, so our return value was an element of the second argument.

```
from sklearn.neighbors import NearestNeighbors

def classify(knownsSamples, knownsOutputs, unknown):
    # knownsSamples: all the data points used for the kneighbors
    # knownsOutputs: a mapping between a knownsSamples and the type of
    # movement. return is an element of this array
    # unknown: what you search for with knn
    # return: search knownsSamples for unknown, get that index, and use
    # that index to find the mapping in the knownsOutputs

    neigh = NearestNeighbors(weights='distance', n_neighbors=1)

    neigh.fit(knownsSamples)

    i = neigh.kneighbors(unknown)[1][0][0]

    return knownsOutputs[i]
```

This looks remarkably similar to the example code, of course, with only minor edits for our specific situation. Although there isn't much code, it is important to consider the parameters, because each small change will heavily skew your data.

3 K Nearest Neighbours in General

One notable aspect of KNN is the fact that it does not actual "learn" or create any model for your data. Rather, it will take the historical data, and match that up with the future data. This is nice because it adapts as you get more data, whereas a different approach, like machine learning, would have required a new model every time data changed. However, you could consider machine learning to be the "smarter" solution. Both have their merits, and it might even have interested implications to repeat the process with a machine learning model.