**Lab Overview**

The lab will focus on applying k-NN classification to the problem of prices prediction. When predicting prices it’s more interesting to look at a dataset where price doesn’t simply increase in proportion to size or the number of characteristics. One domain where this is the case is predicting the prices of wine.

**Topics:**

* Generating a data set
* kNN Price modelling overview
* Defining similarity
* k-NN algorithm
* Weighted neighbours
* Weighted KNN

1. **Generating a data set**

The first thing we need is to get a data set. Normally we gather data from different sources. However, for today's lab we will create a function that artificially simulates a data set.

The **wine\_price/2** function below computes a price for a wine:

(1) based on its age and rating.

(2) using the assumption that wine has a peak age, which is:

- older for good wines

- and almost immediate for bad wines.

|  |
| --- |
| from random import random, randint  import math  def wine\_price(rating,age):  peak\_age = rating - 50    # Calculate price based on rating  price = rating/2  if age > peak\_age:  # Past its peak, goes bad in 10 years  price = price \* (5 - (age - peak\_age)/2)  else:  # Increases to 5x original value as it approaches its peak  price = price \* (5 \* ((age + 1) / peak\_age))  if price < 0: price = 0  return price |

Test the wine\_price() function:

> wine\_price(95.0,3.0)

21.111111111111114

> wine\_price(95.0,8.0)

47.5

> wine\_price(99.0,1.0)

10.102040816326529

We can use the wine\_price() function to build of data set of wines.

The wine\_set() function generates 300 random bottles of wine and calculates their prices using the wine\_price() function above.

It then randomly adds or subtracts 20 percent to capture things like taxes and local variations in prices, and also to make the numerical prediction a bit more difficult.

|  |
| --- |
| def wine\_set():  rows = []  for i in range(300):  # Create a random age and rating  rating = random()\*50 + 50  age = random()\*50  # Get reference price  price = wine\_price(rating,age)    # Add some noise  price \*= random()\*0.2 + 0.9  # Add to the dataset  rows.append({'input':(rating,age), 'result':price})    return rows |

Now let’s use the wine\_set() function to generate a dataset.

Remember that we use a random function in the dataset generation, so the data set you generate will be different from the one used when creating the examples. So you will see slightly different numbers in the examples.

> data = wine\_set( )

> data[0]

{'input': (63.602840187200407, 21.574120872184949), 'result': 34.565257353086487}

> data[1]

{'input': (74.994980945756794, 48.052051269308649), 'result': 0.0}

**k-NN Algorithm**

k-NN Prices Modelling Approach finds a few of the most similar items and assumes the prices will be roughly the same.

By finding a set of items similar to the item that interests you, the algorithm can average their prices and make a guess at what the price should be for this item.

If the data were perfect, you could use k=1, meaning you would just pick the nearest neighbour and use its price as the answer. But there are always aberrations, so it is best to average over a few neighbours to reduce any noise.

Selecting k is difficult: to low and the prediction is overly sensitive to noise, to high and the prediction is affected by examples that are not similar.

1. **Defining Similarity**

There are many ways to define similarity. **Euclidean distance** is one of the most popular ones.

|  |
| --- |
| def euclidean(instance1, instance2):  dist = 0.0  for i in range(len(instance1)):  dist += (instance1[i] - instance2[i])\*\*2    return math.sqrt(dist) |

*Note*: this function treats both age and rating the same when calculating distance, even though in almost any real problem, some of variables have a greater impact on the final price than others. This is a well-known weakness of kNN.

Try the function on some of the points in your dataset, along with a new data point:

>data[0]['input']

(82.720398223643514, 49.21295829683897)

> data[1]['input']

(98.942698715228076, 25.702723509372749)

> euclidean(data[0]['input'], data[1]['input'])

28.56386131112269

In the lecture we talked about other direct similarity metrics such as the Manhattan distance and osine.

**Task:** Implement functions to compute Manhattan distance.Note, you should use methods from the Python mathmodule to define these functions.

Define a method to that implements the Manhattan Distance. Example interactions with this function are:

> manhattan([1,1],[1,0])

1.0

> manhattan([0,1],[1,0])

2.0

1. **Implement k-NN Algorithm**

Implementing the k-NN algorithm is simple. We need two functions:

* **get\_distances**/2 function computes the distances between a query and the examples in the dataset
* **knn\_estimate**/3 function averages over the nearest neighbours

The getdistances/2 function calls the distance function on the vector given against every other vector in the dataset and puts them in a big list. The list is sorted so that the closest item is at the top.

|  |
| --- |
| def get\_distances(data, query):  distance\_list = []    # Loop over every item in the dataset  for i in range(len(data)):  instance = data[i]['input']    # Add the distance and the index  distance\_list.append((euclidean(instance,query),i))    # Sort by distance  distance\_list.sort()    return distance\_list |

The k-NN function uses the list of distances and averages the top k results

|  |
| --- |
| def knn\_estimate(data, query, k =5):  # Get sorted distances  dlist = get\_distances(data,query)  avg = 0.0    # Take the average of the top k results  for i in range(k):  idx = dlist[i][1]  avg += data[idx]['result']    avg = avg/k  return avg |

You can now get a price estimate for a new item:

> knn\_estimate(data,(95.0,3.0))

29.176138546872018

> knn\_estimate(data,(99.0,5.0))

37.610888778473793

> wine\_price(99.0, 5.0) # Get the actual price

30.306122448979593

*# Try with fewer neighbours*

> knn\_estimate(data, (99.0,5.0), k=1)

38.078819347238685

**Task:** Try different parameters and different values for k to see how the results are affected

1. **Weighted Neighbours**

One way to compensate for the fact that the algorithm may be using neighbours that are too far away is to weight them according to their distance. Therefore we need a way of converting distances to weights.

**Inverse Function**

Simplest form returns 1 divided by the distance. However, if items are exactly the same will result in infinite weight. We can avoid this by adding a small number to the distance before inverting it.

|  |
| --- |
| def inverse\_weight(dist, const = 0.1):  return 1.0/(dist + const) |

*Pros*: simple to implement and fast

*Cons*: applies very heavy weights to items close by and falls of quickly after that. This can make the algorithm sensitive to noise.

**Subtract Function**

Subtract the distance from a constant. The weight is:

(1) the result of this subtraction if the result is greater than zero;

(2) otherwise, the result is zero.

|  |
| --- |
| def subtract\_weight(dist, const = 1.0):  if dist > const:  return 0  else:  return const - dist |

*Pros*: Simple to implement and fast. Overcomes the issue of overweighting close items.

*Cons*: Because the weight eventually falls to 0, it’s possible that there will be nothing close enough to be considered a close neighbour, which means that for some items the algorithm won’t make a prediction all.

**Python Note**

Python allows you to pass methods as argument to other methods. For example,

|  |
| --- |
| def method1():  return 'hello world'  def method2(methodToRun):  result = methodToRun()  return result  method2(method1) |

**Task**: Define a function that takes a weight method as an argument and applies it to an array of number. For example, assuming the function you defined is called apply\_weight() you should be able to use it as follows:

> distances = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]

> apply\_weigth(inverse\_weight, distances)

[5.0, 3.333333333333333, 2.5, 2.0, 1.6666666666666667, 1.4285714285714286, 1.25, 1.1111111111111112, 1.0, 0.9090909090909091]

**Weighted k-NN**

The code for doing weighted kNN works the same way as the regular kNN function by:

(1) getting the sorted distances

(2) and taking the k closest elements.

The important difference is that instead of just averaging them, the weighted kNN calculates a weighted average.

The weighted average is calculated by:

(1) multiplying each item’s weight by its value

(2) summing the results of these multiplications before adding them together

(3) dividing the sum from (2) by the sum of all the weights.

|  |
| --- |
| def weighted\_knn(data, vec1, k=5, weightf=inverse\_weight):  # Get distancesabs  dist\_list = get\_distances(data,vec1)  avg = 0.0  totalweight = 0.0    # Get weighted average  for i in range(k):  dist = dist\_list[i][0]  idx = dist\_list[i][1]  weight = weightf(dist)  avg += weight\*data[idx]['result']  totalweight += weight    if totalweight == 0: return 0    avg = avg/totalweight  return avg |

The function loops over the k nearest neighbours and passes each of their distances to one of the weight functions you defined earlier.

The avg variable is calculated by multiplying these weights by the neighbour’s value. The totalweight variable is the sum of the weights. At the end, avg is divided by totalweight.

**Task**: Try this function in our python session and compare its performance to that of the regular kNN function:

> weighted\_knn(data,(99.0,5.0))

32.640981119354301