KNN and K-means clustering

Computational Complexity

Linear time
Polynomial time
Non-polynomial time (exponential or worse)

Constant, linear and superlinear time

O(1)

- Append to a list

O(log (n))

- Search in a sorted sequence

O(n)

- Count events, compare sequences

O(n log(n))

- FFT

Polynomial time

O(n^2)

- All-pairs distance

O(n^3)

- Matrix multiplication

O (n^4)

- CT – reconstruktion

Non-polynomial time

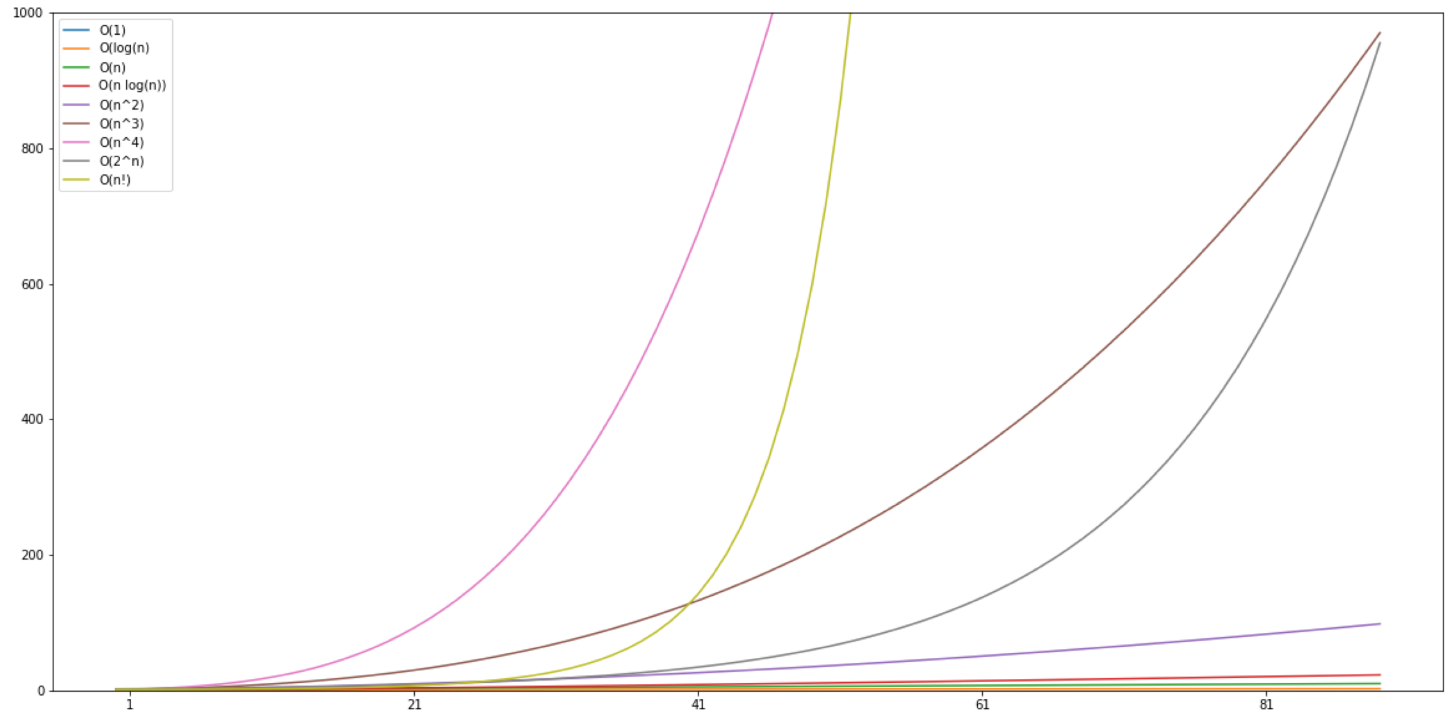
O(2^n)

- Knapsack problem

O(n!)

- N-droning problem

Choices has consequences



Quiz 1

Consider this;

1. How long time does a CPU take to execute a calculation?
2. How long time does a CPU take to read a word from memory?
3. How long time to read a word from a persistant storage?

What is the consequence of the above answers in a Big Data setting?

KNN

K Nearest neighbors

We have an annotated database and we wish to classify new data-points using that database

The fundamental assumption is that a point is similar to its neighbors

We can thus guess a class by looking a a set of neighboring points

KNN is often used as a quick and dirty test to see if machine learning will work

“WE ARE ALL INDIVIDUALS!”
“WE ARE ALL DIFFERENT!”
“...I’m not.”

K Nearest Neighbors

Simple election

Find the K points nearest in space

Make an election between them

Classify as the same as the most common neighbor

Algorithm

To classify a new point:

Find distance to all other points

Find the k lowest distances in that list

Select the most common type amongst the k closest as the class of the unknown

Measuring the distance is non-trivial as it is a means of adapting the KNN algorithm to a problem.

In this talk we will normalize all parameters to be in the interval [0:1] and use Euclidian distance

- It is still hard as dimensionality grows

Complexity

$$O(n)$$

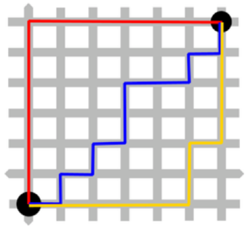
Distances

Usually just Euclidian distance

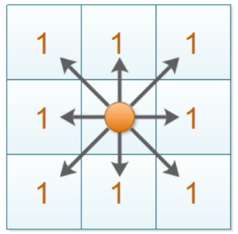
$$d(\mathbf{x}_i, \mathbf{x}_l) = \sqrt{(x_{i1} - x_{l1})^2 + (x_{i2} - x_{l2})^2 + \cdots + (x_{ip} - x_{lp})^2}.$$

Other common distances

Manhattan



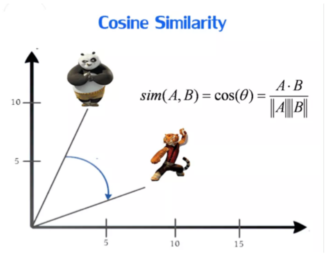
Chebyshev



Canberra

$$d^{CAD}(i,j)=\sum_{k=0}^{n-1}\frac{|y_{i,k}-y_{j,k}|}{|y_{i,k}|+|y_{j,k}|}$$

Cosine



Library exist

import scipy.spatial.distance as dist

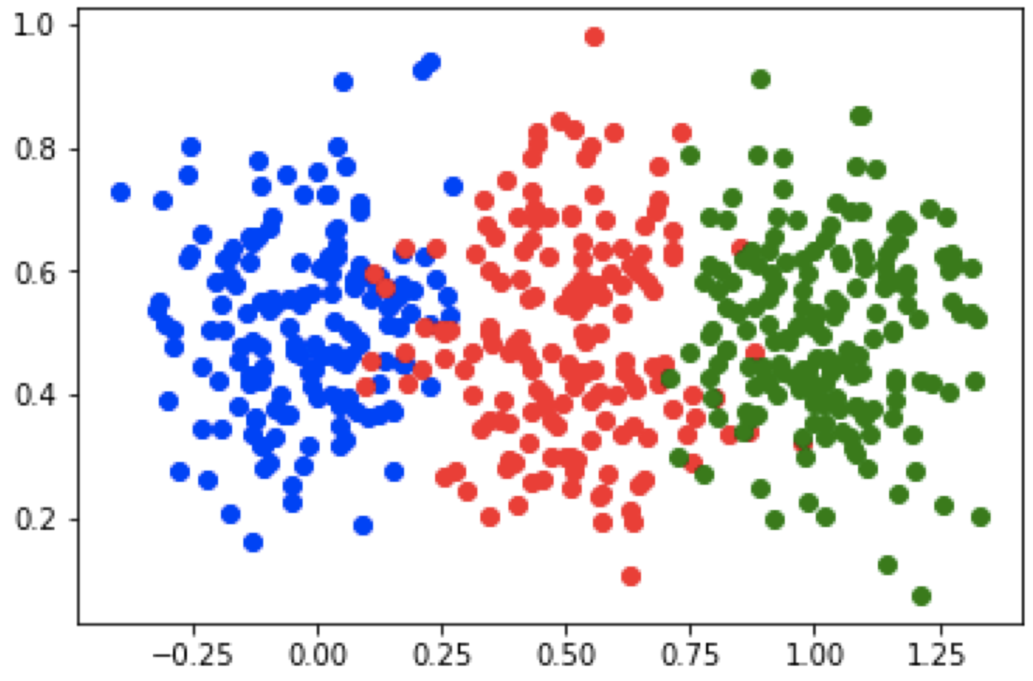
A first artificial example

Huey, Dewey and Louie

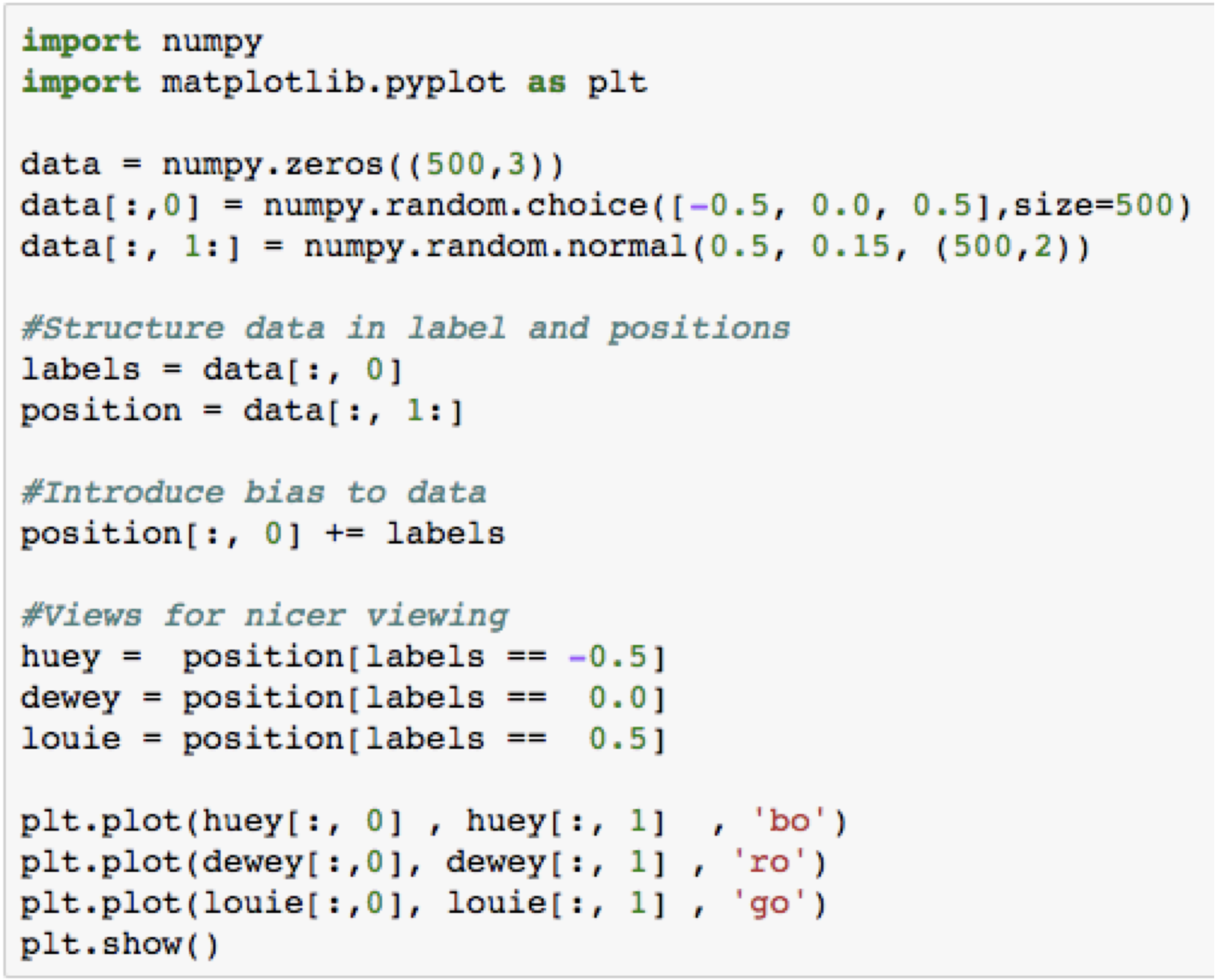
Image borrowed from: http://disney.wikia.com/wiki/Huey,_Dewey_and_Louie

K Nearest Neighbors

Huey, Dewey and Louie share a room, but has an affinity to stay around their bed.



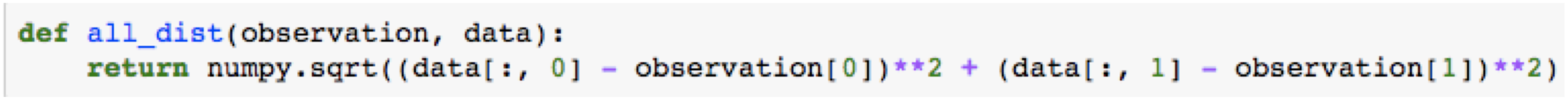
Generate artificial data



```python
import numpy
import matplotlib.pyplot as plt

data = numpy.zeros((500,3))
data[:,0] = numpy.random.choice([-0.5, 0.0, 0.5],size=500)
data[:, 1:] = numpy.random.normal(0.5, 0.15, (500,2))

#Structure data in label and positions
labels = data[:, 0]
position = data[:, 1:]

#Introduce bias to data
position[:, 0] += labels

#Views for nicer viewing
huey =  position[labels == -0.5]
dewey = position[labels ==  0.0]
louie = position[labels ==  0.5]

plt.plot(huey[:, 0] , huey[:, 1]  , 'bo')
plt.plot(dewey[:,0], dewey[:, 1] , 'ro')
plt.plot(louie[:,0], louie[:, 1] , 'go')
plt.show()
```

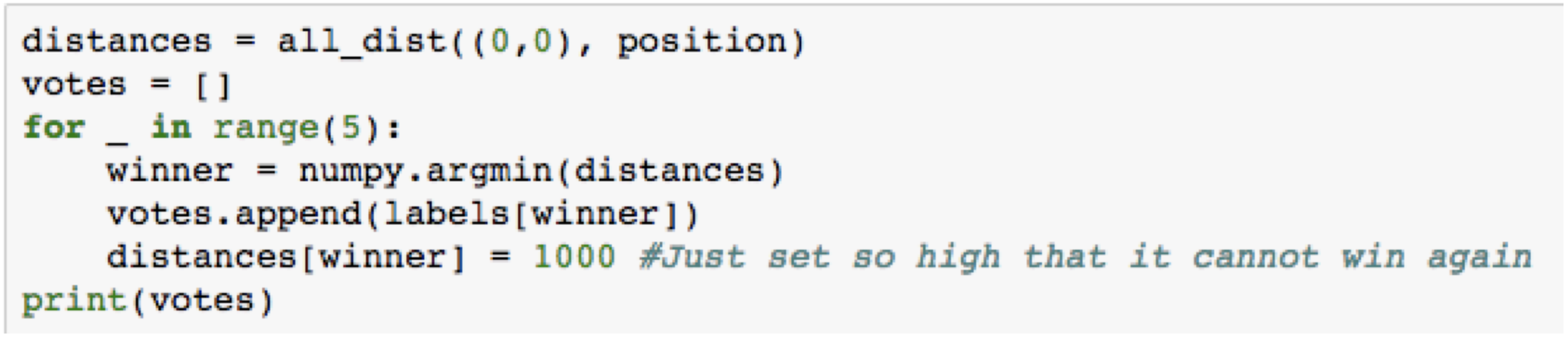
Distance

We are already in range [0:1] so easy



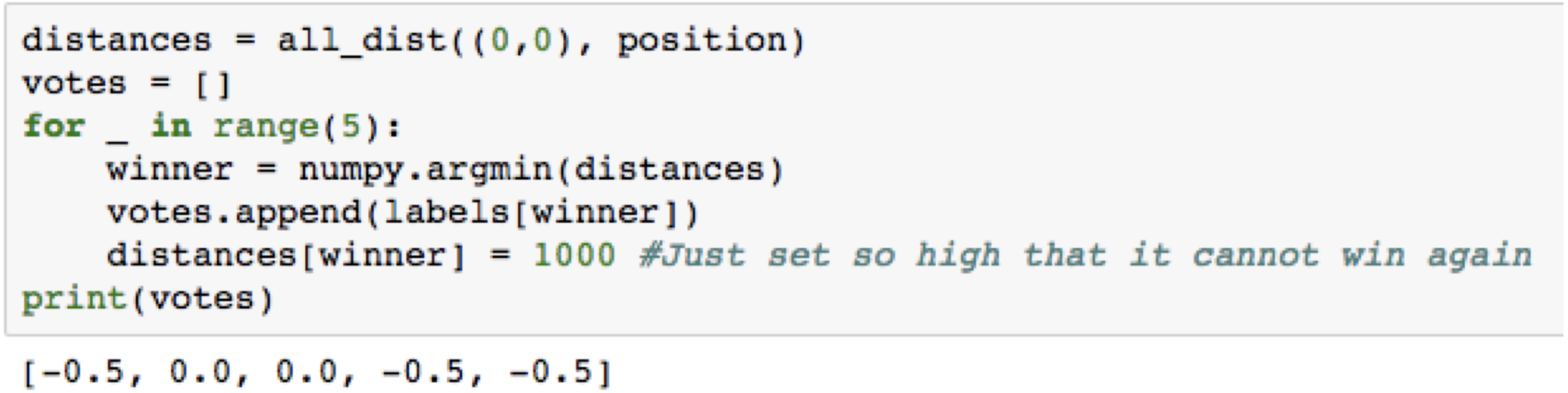
```python
def all_dist(observation, data):
    return numpy.sqrt((data[:, 0] - observation[0])**2 + (data[:, 1] - observation[1])**2)
```

Find the closest neighbors



```
distances = all_dist((0,0), position)
votes = []
for _ in range(5):
    winner = numpy.argmin(distances)
    votes.append(labels[winner])
    distances[winner] = 1000 #Just set so high that it cannot win again
print(votes)
```

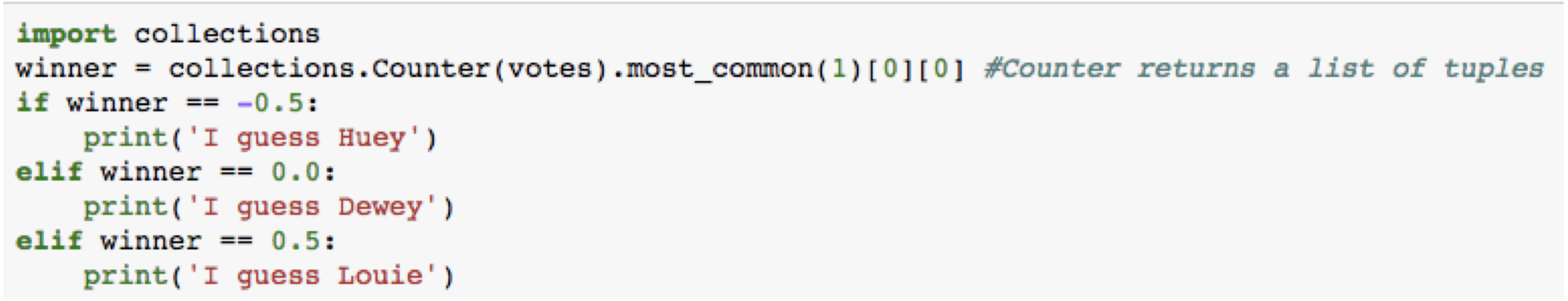
Find the closest neighbors



```
distances = all_dist((0,0), position)
votes = []
for _ in range(5):
    winner = numpy.argmin(distances)
    votes.append(labels[winner])
    distances[winner] = 1000 #Just set so high that it cannot win again
print(votes)
```

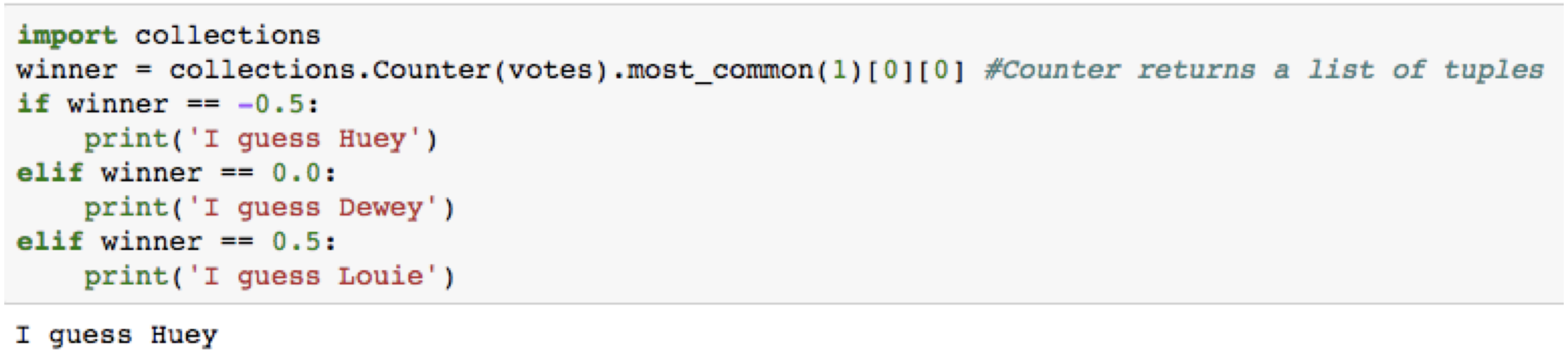
```
[-0.5, 0.0, 0.0, -0.5, -0.5]
```

Tally the votes



```python
import collections
winner = collections.Counter(votes).most_common(1)[0][0] #Counter returns a list of tuples
if winner == -0.5:
    print('I guess Huey')
elif winner == 0.0:
    print('I guess Dewey')
elif winner == 0.5:
    print('I guess Louie')
```

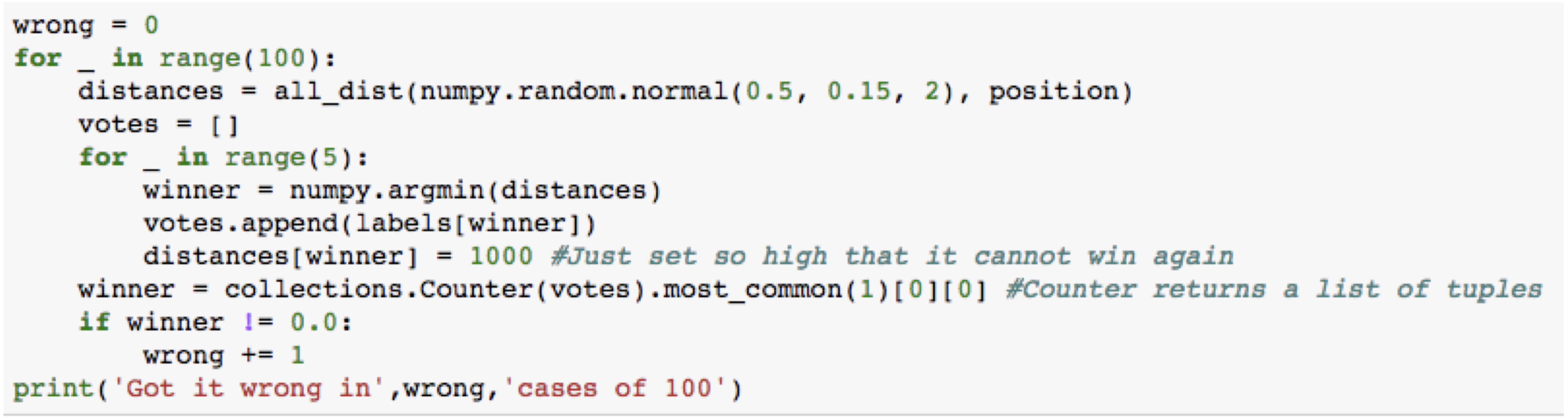
Tally the votes



```python
import collections
winner = collections.Counter(votes).most_common(1)[0][0] #Counter returns a list of tuples
if winner == -0.5:
    print('I guess Huey')
elif winner == 0.0:
    print('I guess Dewey')
elif winner == 0.5:
    print('I guess Louie')
```

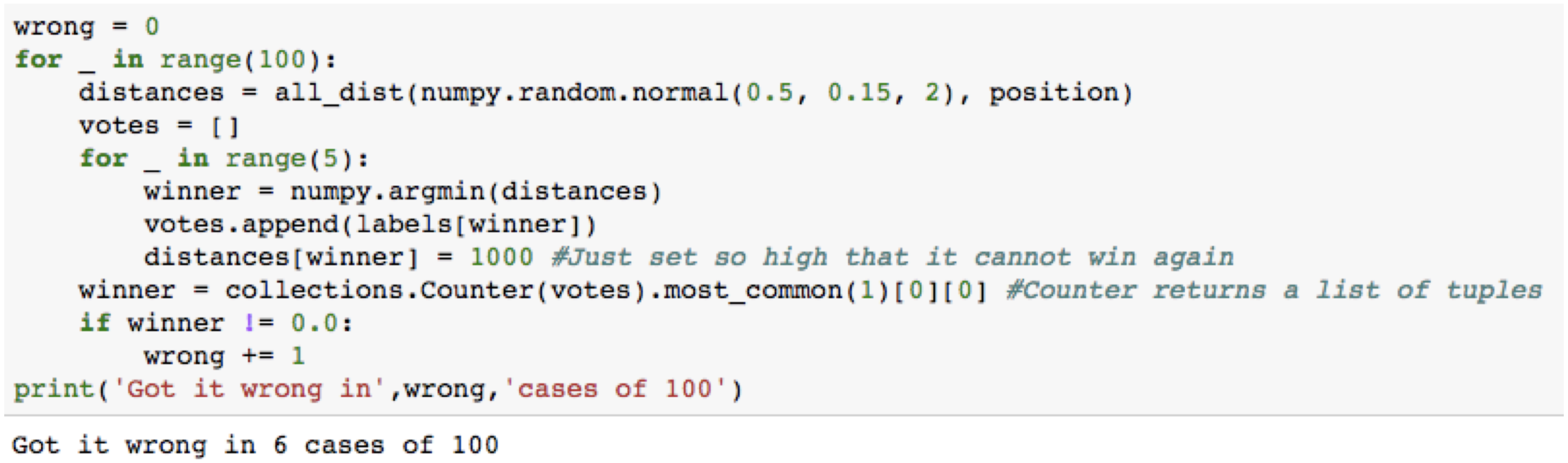
```
I guess Huey
```

Is it robust?



```python
wrong = 0
for _ in range(100):
    distances = all_dist(numpy.random.normal(0.5, 0.15, 2), position)
    votes = []
    for _ in range(5):
        winner = numpy.argmin(distances)
        votes.append(labels[winner])
        distances[winner] = 1000 #Just set so high that it cannot win again
    winner = collections.Counter(votes).most_common(1)[0][0] #Counter returns a list of tuples
    if winner != 0.0:
        wrong += 1
print('Got it wrong in',wrong,'cases of 100')
```

Is it robust?



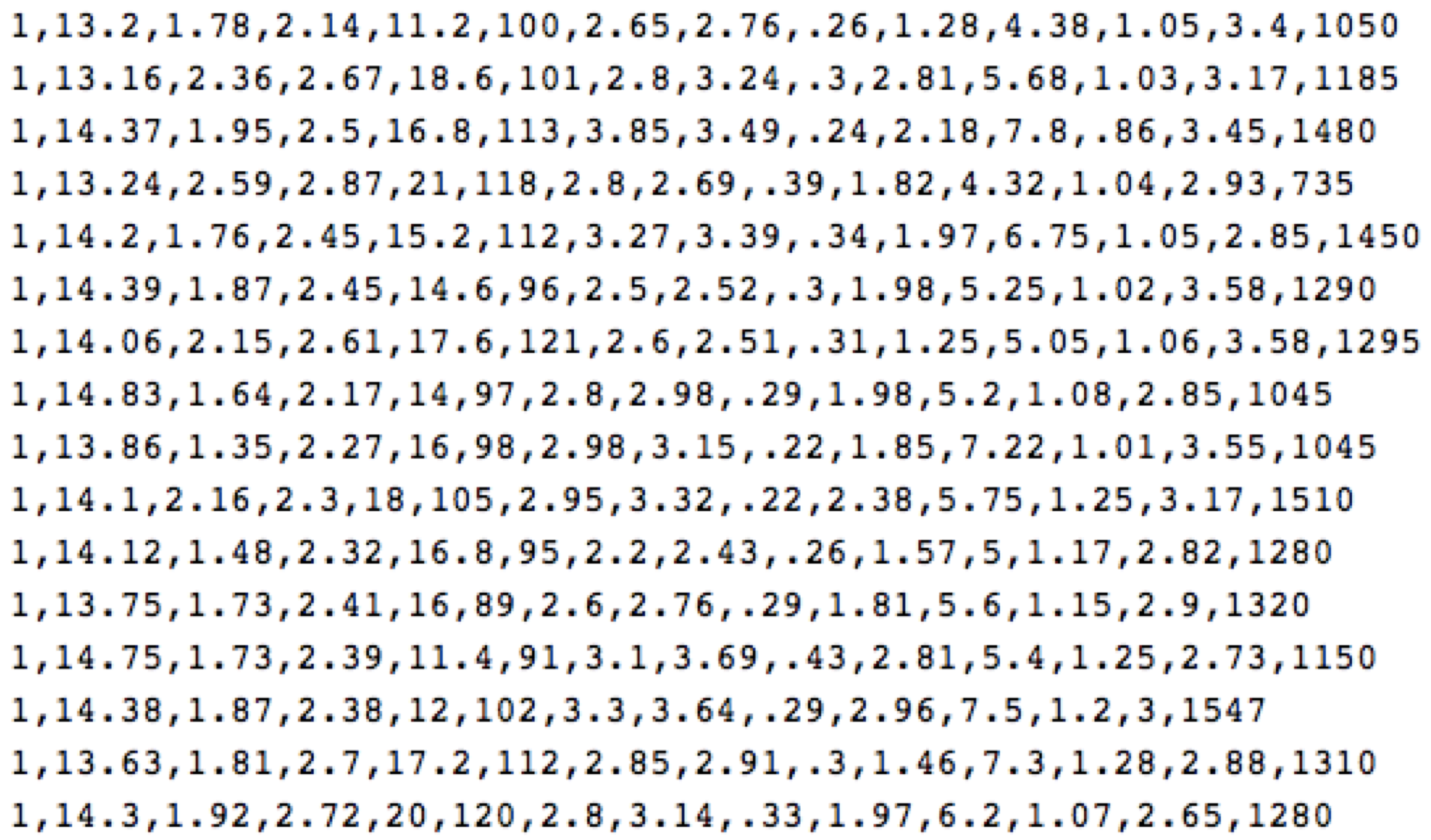
```python
wrong = 0
for _ in range(100):
    distances = all_dist(numpy.random.normal(0.5, 0.15, 2), position)
    votes = []
    for _ in range(5):
        winner = numpy.argmin(distances)
        votes.append(labels[winner])
        distances[winner] = 1000 #Just set so high that it cannot win again
    winner = collections.Counter(votes).most_common(1)[0][0] #Counter returns a list of tuples
    if winner != 0.0:
        wrong += 1
print('Got it wrong in',wrong,'cases of 100')
```

```
Got it wrong in 6 cases of 100
```

A larger example - Wine classifier

1) Alcohol
2) Malic acid
3) Ash
4) Alcalinity of ash
5) Magnesium
6) Total phenols
7) Flavanoids
8) Nonflavanoid phenols
9) Proanthocyanins
10) Color intensity
11) Hue
12) OD280/OD315 of diluted wines
13) Proline

A larger example - Wine classifier



```
1,13.2,1.78,2.14,11.2,100,2.65,2.76,.26,1.28,4.38,1.05,3.4,1050
1,13.16,2.36,2.67,18.6,101,2.8,3.24,.3,2.81,5.68,1.03,3.17,1185
1,14.37,1.95,2.5,16.8,113,3.85,3.49,.24,2.18,7.8,.86,3.45,1480
1,13.24,2.59,2.87,21,118,2.8,2.69,.39,1.82,4.32,1.04,2.93,735
1,14.2,1.76,2.45,15.2,112,3.27,3.39,.34,1.97,6.75,1.05,2.85,1450
1,14.39,1.87,2.45,14.6,96,2.5,2.52,.3,1.98,5.25,1.02,3.58,1290
1,14.06,2.15,2.61,17.6,121,2.6,2.51,.31,1.25,5.05,1.06,3.58,1295
1,14.83,1.64,2.17,14,97,2.8,2.98,.29,1.98,5.2,1.08,2.85,1045
1,13.86,1.35,2.27,16,98,2.98,3.15,.22,1.85,7.22,1.01,3.55,1045
1,14.1,2.16,2.3,18,105,2.95,3.32,.22,2.38,5.75,1.25,3.17,1510
1,14.12,1.48,2.32,16.8,95,2.2,2.43,.26,1.57,5,1.17,2.82,1280
1,13.75,1.73,2.41,16,89,2.6,2.76,.29,1.81,5.6,1.15,2.9,1320
1,14.75,1.73,2.39,11.4,91,3.1,3.69,.43,2.81,5.4,1.25,2.73,1150
1,14.38,1.87,2.38,12,102,3.3,3.64,.29,2.96,7.5,1.2,3,1547
1,13.63,1.81,2.7,17.2,112,2.85,2.91,.3,1.46,7.3,1.28,2.88,1310
1,14.3,1.92,2.72,20,120,2.8,3.14,.33,1.97,6.2,1.07,2.65,1280
```

Read the data

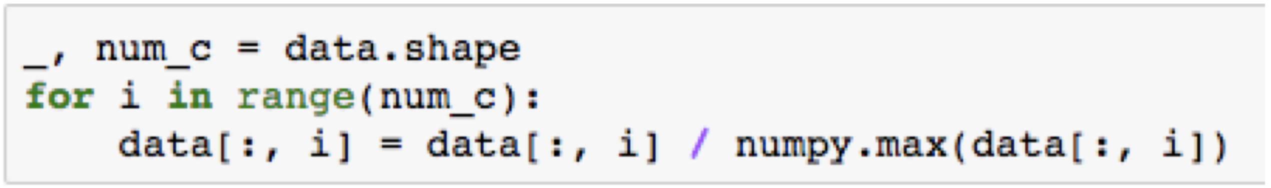
```
import csv
import numpy

with open('wine.data') as input_file:
    raw_data = numpy.array([row for row in csv.reader(input_file)]).astype(numpy.float)

labels = raw_data[:, 0 ]
data   = raw_data[:, 1:]
```

Normalization

Since the data for the different columns differ enormously we need to scale to a common space

- The easy and common approach is to scale to [0:1]

Normalize

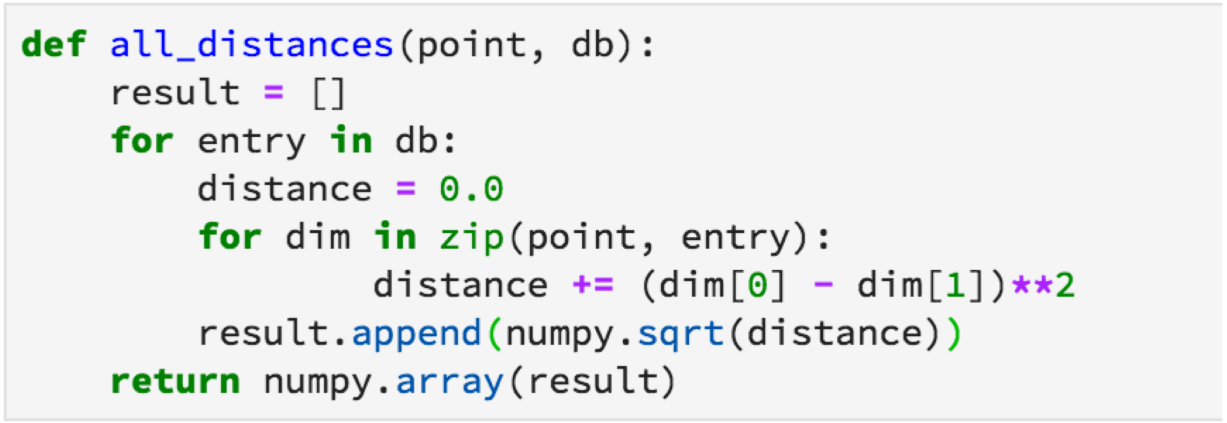


```
_, num_c = data.shape
for i in range(num_c):
    data[:, i] = data[:, i] / numpy.max(data[:, i])
```

Distances

Here we have several dimensions so it is easier to write a generic distance function that is indifferent to dimensions

Distance function

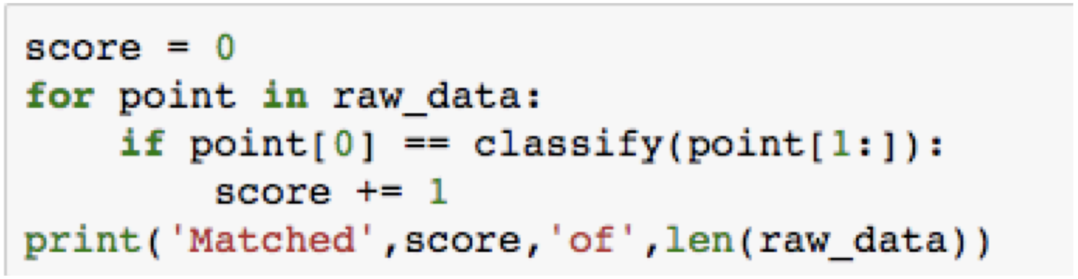


```python
def all_distances(point, db):
    result = []
    for entry in db:
        distance = 0.0
        for dim in zip(point, entry):
                distance += (dim[0] - dim[1])**2
        result.append(numpy.sqrt(distance))
    return numpy.array(result)
```

Classifier

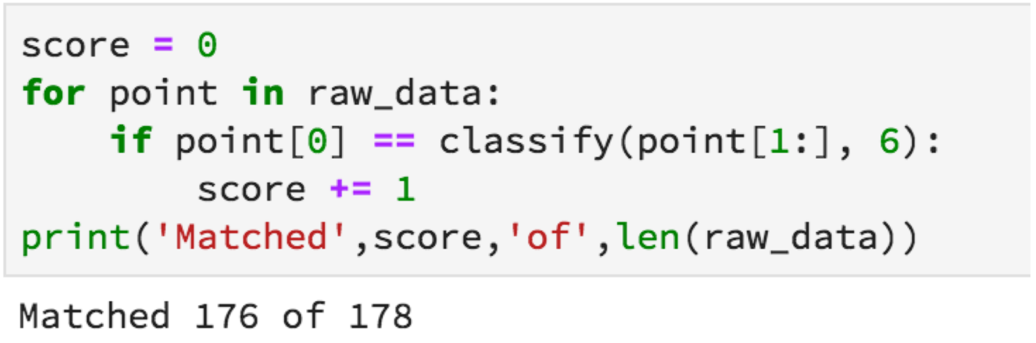
```
import collections
def classify(point, k=5):
    distances = all_distances(point, data)
    votes = []
    for _ in range(k):
        winner = numpy.argmin(distances)
        votes.append(labels[winner])
        distances[winner] = 1000
    return collections.Counter(votes).most_common(1)[0][0]
```

Test



```python
score = 0
for point in raw_data:
    if point[0] == classify(point[1:]):
        score += 1
print('Matched',score,'of',len(raw_data))
```

Test



```
score = 0
for point in raw_data:
    if point[0] == classify(point[1:], 6):
        score += 1
print('Matched',score,'of',len(raw_data))
```

```
Matched 176 of 178
```

Exercise

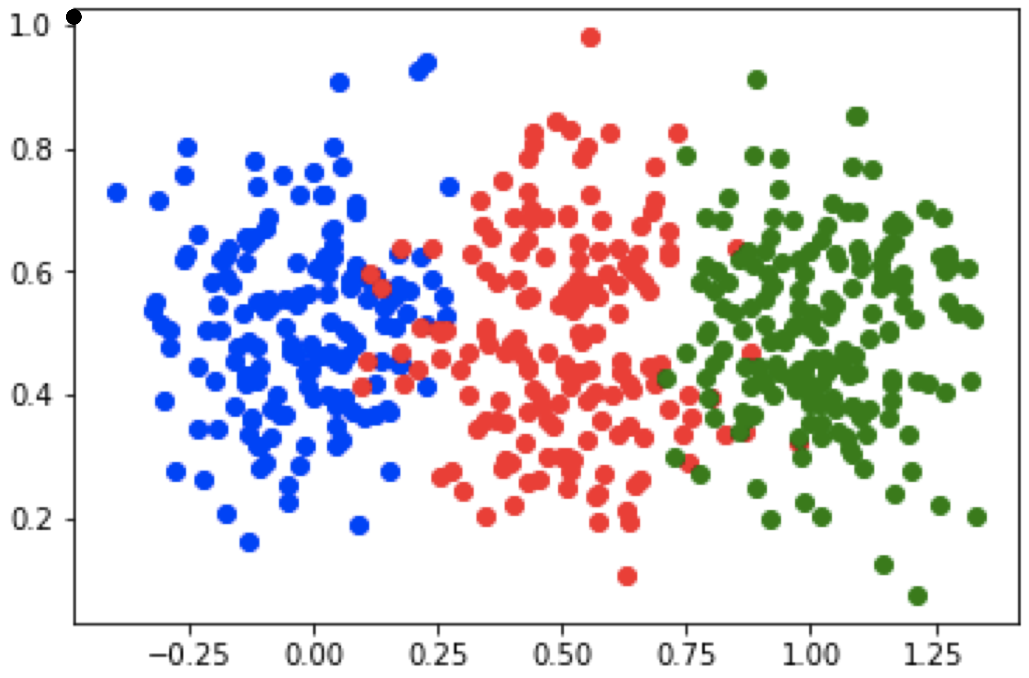
The result is quite satisfactory. However, since we are matching against the database itself, the tested point is itself in the test set, which is an unfair advantage compared to a real world scenario. Eliminating this bias is left as an exercise, it is quite simple though.

Play with different values of K to see if 5 is indeed the best choice.

Other uses

Outlier detection

Use a simple static measure, stdev, on the sum of distance to neighbors



Quiz 2

How does KNN match the problem you are working on in this class?

K-means Clustering

Clustering

Clusters are sets of data points that we deem to be related

- Typically this means that they are of the same class

For clustering we do not have any ground-truth

We often know how many clusters we will need though

K-means clustering

Very related to k nearest neighbors

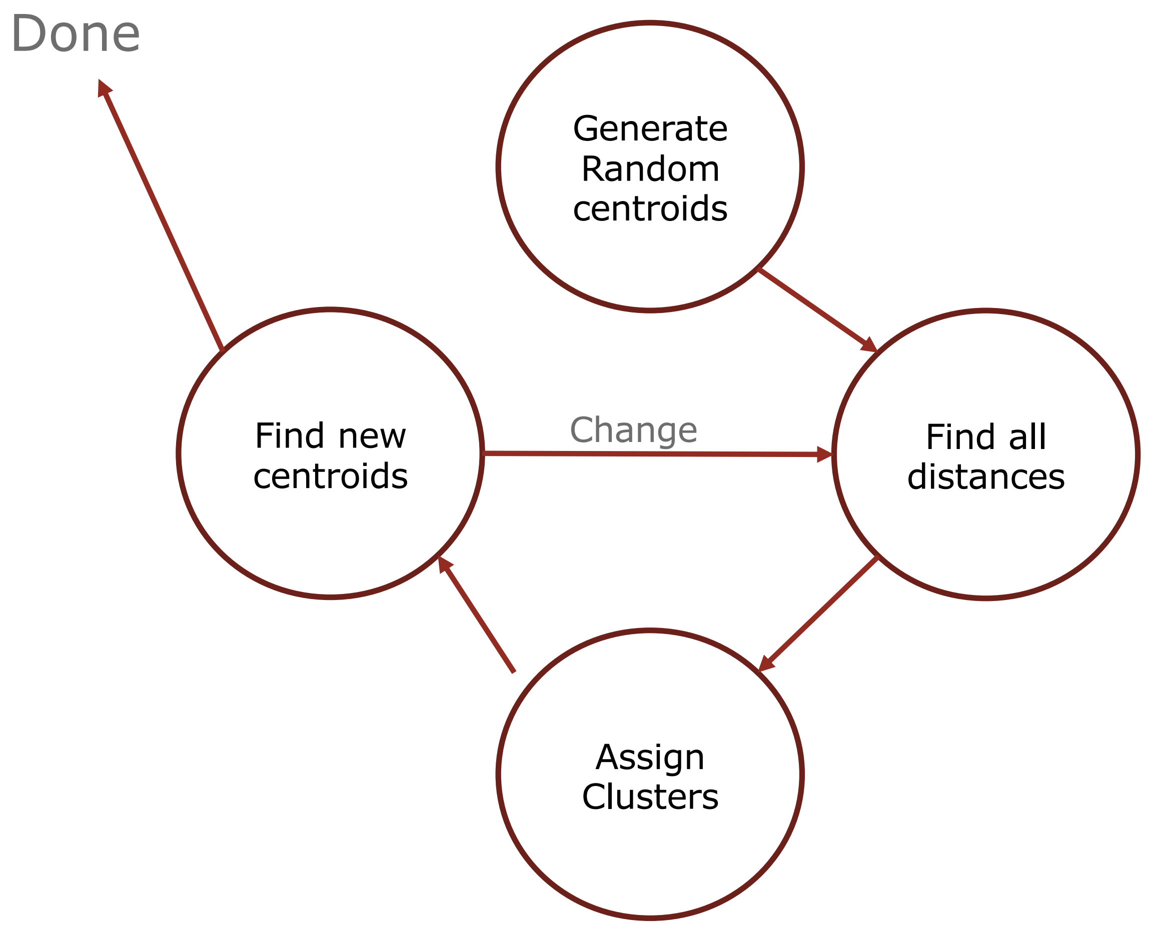
Used when we have a dataset where we don’t know the classes but need to classify elements

For now we will assume we know the number of clusters we are looking for

The simple Algorithm

1. Choose k random points in space as centroids
2. Find the distance from each data-point to each centroid
3. Assign data points to the centroid they are closest to
4. Move the centroids to the center of the points associated with it
5. If centroids changed repeat from 2

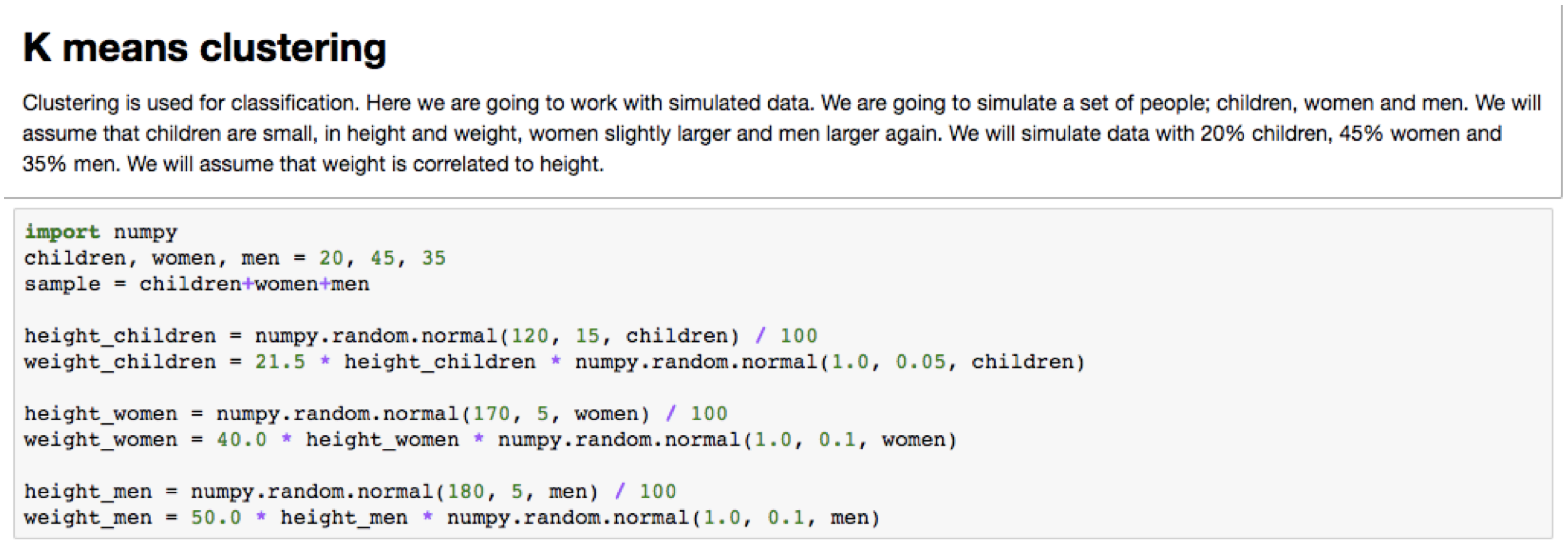
Flow


Done
Generate Random centroids
Find all distances
Assign Clusters
Find new centroids
Change

Complexity

$$O(nkl)$$

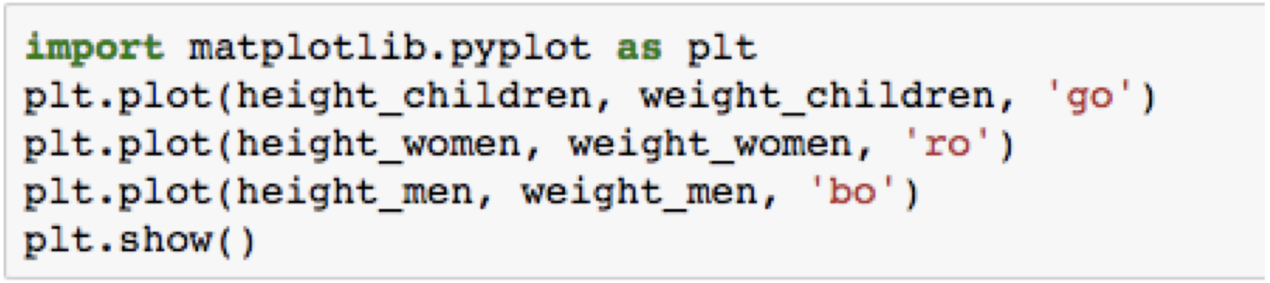
Artificial example



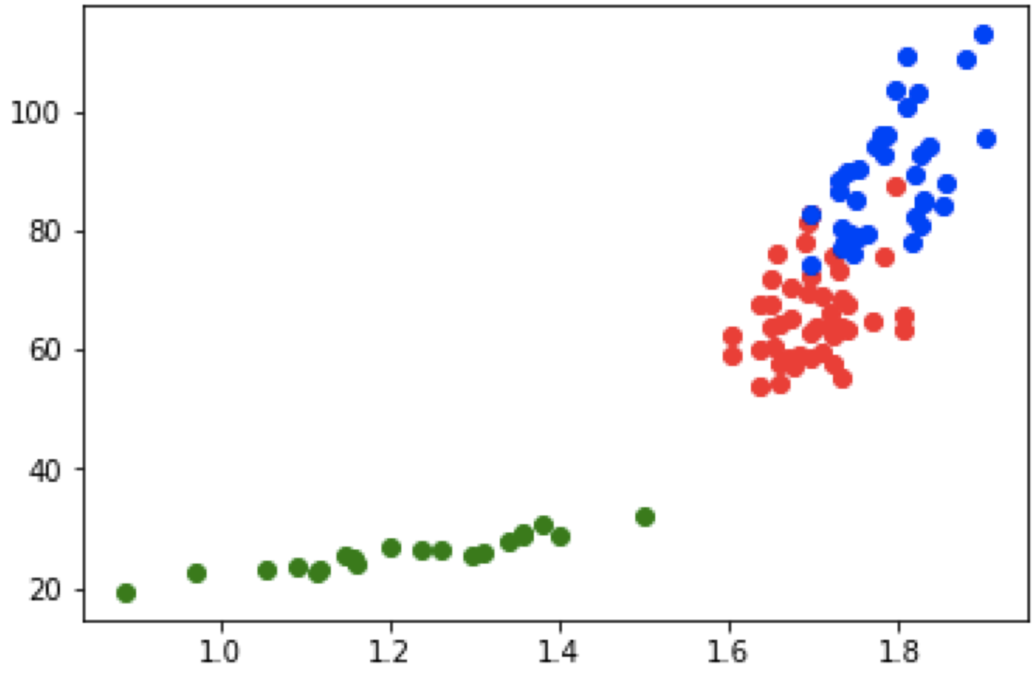
K means clustering

Clustering is used for classification. Here we are going to work with simulated data. We are going to simulate a set of people; children, women and men. We will assume that children are small, in height and weight, women slightly larger and men larger again. We will simulate data with 20% children, 45% women and 35% men. We will assume that weight is correlated to height.

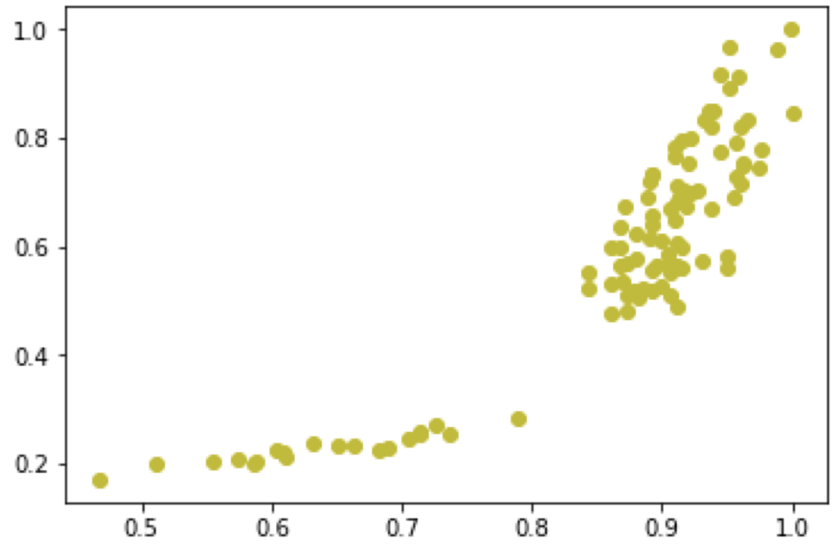
```python
import numpy
children, women, men = 20, 45, 35
sample = children+women+men

height_children = numpy.random.normal(120, 15, children) / 100
weight_children = 21.5 * height_children * numpy.random.normal(1.0, 0.05, children)

height_women = numpy.random.normal(170, 5, women) / 100
weight_women = 40.0 * height_women * numpy.random.normal(1.0, 0.1, women)

height_men = numpy.random.normal(180, 5, men) / 100
weight_men = 50.0 * height_men * numpy.random.normal(1.0, 0.1, men)
```



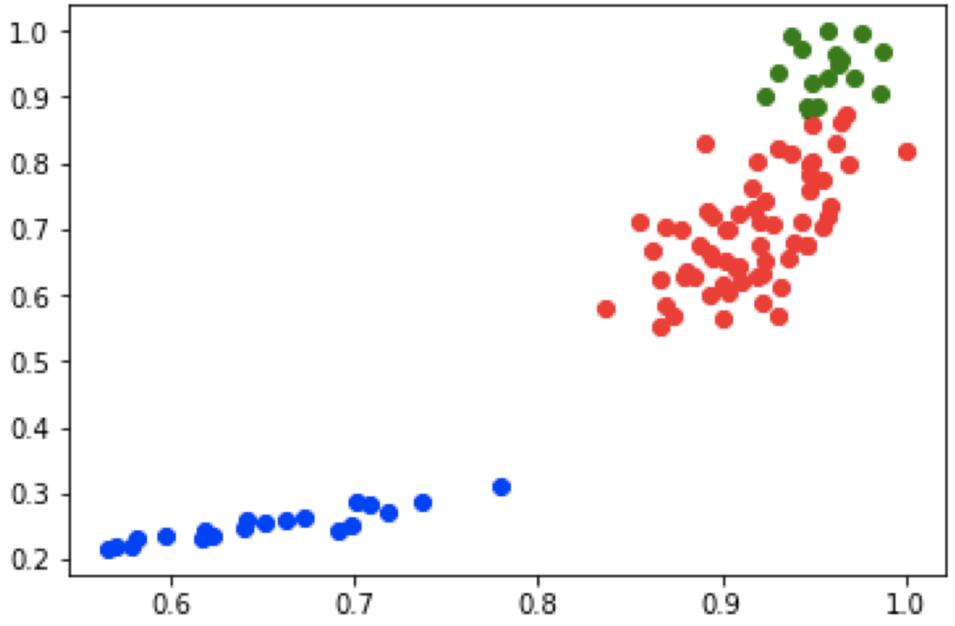
```
import matplotlib.pyplot as plt
plt.plot(height_children, weight_children, 'go')
plt.plot(height_women, weight_women, 'ro')
plt.plot(height_men, weight_men, 'bo')
plt.show()
```



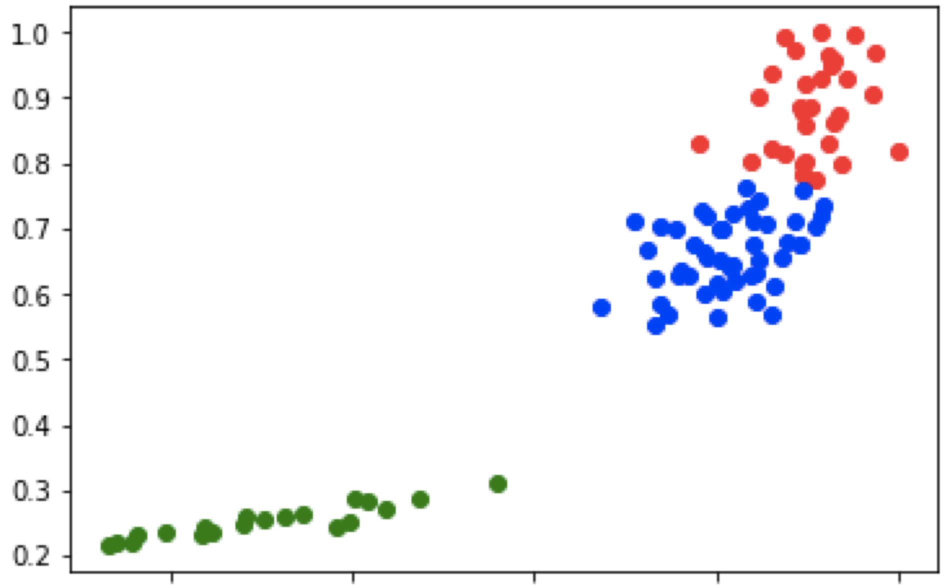
```
raw_data = numpy.concatenate((numpy.array((height_children, weight_children)), \
                              numpy.array((height_women, weight_women)) , \
                              numpy.array((height_men, weight_men))), axis = 1)

data = raw_data / numpy.max(raw_data, axis = 1)[numpy.newaxis].T #We transpose to have the data in rows
plt.plot(data[0, :], data[1, :], 'yo')
plt.show()
```

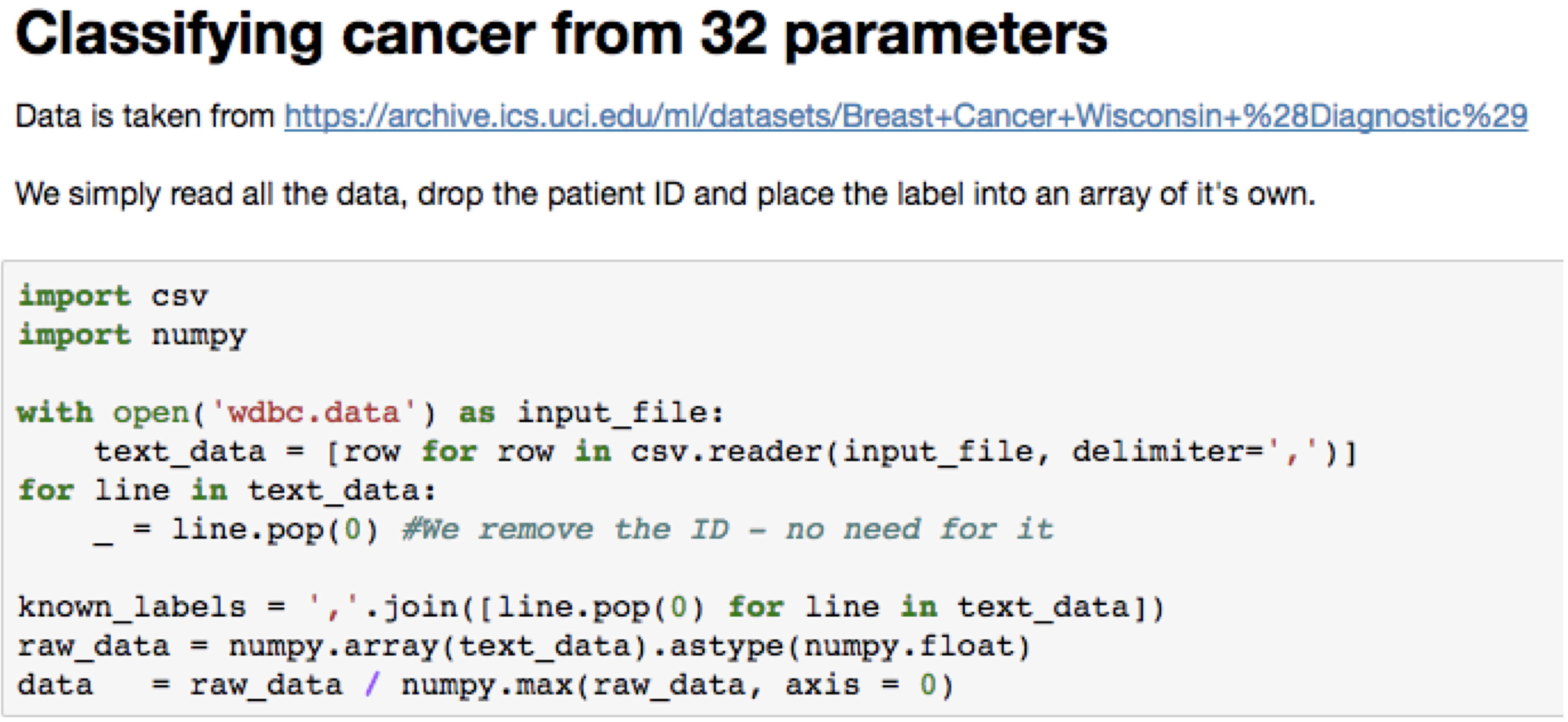


```
def all_dist(observation, data):
    return numpy.sqrt((data[0, :] - observation[0])**2 + (data[1, :] - observation[1])**2)

k=3
centroids = numpy.array([data[:, numpy.random.randint(sample)] for _ in range(k)])
distances = numpy.empty((k,sample))
for d in range(k):
    distances[d, :] = all_dist(centroids[d], data)
winners = numpy.argmin(distances, axis = 0)
clusters = [data[:, winners == i] for i in range(k)]
for cluster, color in zip(clusters, ['go', 'ro', 'bo']):
    plt.plot(cluster[0, :], cluster[1, :], color)
plt.show()
```



```python
def cluster(data, k):
    centroids = numpy.array([data[:, numpy.random.randint(sample)] for _ in range(k)])
    done = False
    while not done:
        distances = numpy.empty((k,sample))
        for d in range(k):
            distances[d, :] = all_dist(centroids[d], data)
        winners = numpy.argmin(distances, axis = 0)
        clusters = [data[:, winners == i] for i in range(k)]
        prev_centroids = centroids
        centroids = numpy.array([numpy.average(cluster, axis = 1) for cluster in clusters])
        if numpy.sum(prev_centroids-centroids) == 0:
            done=True
    for cluster, color in zip(clusters, ['go', 'ro', 'bo']):
        plt.plot(cluster[0, :], cluster[1, :], color)
    plt.show()

cluster(data,3)
```



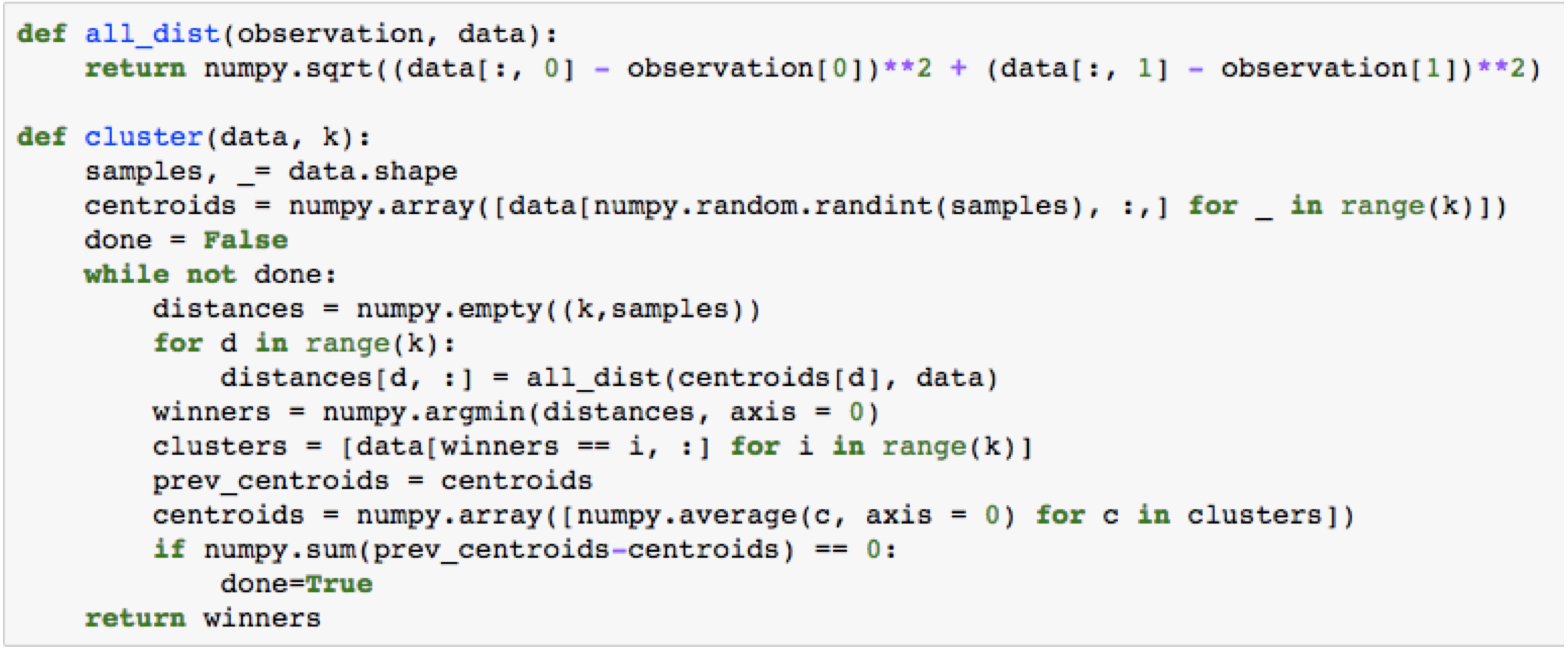


Classifying cancer from 32 parameters

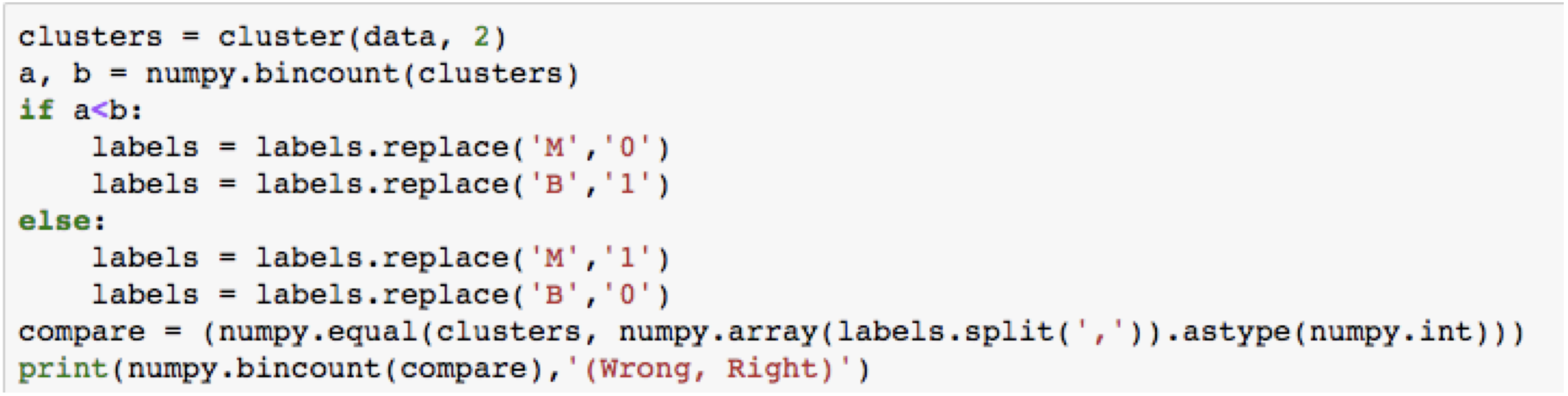
Data is taken from https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+%28Diagnostic%29

We simply read all the data, drop the patient ID and place the label into an array of it's own.

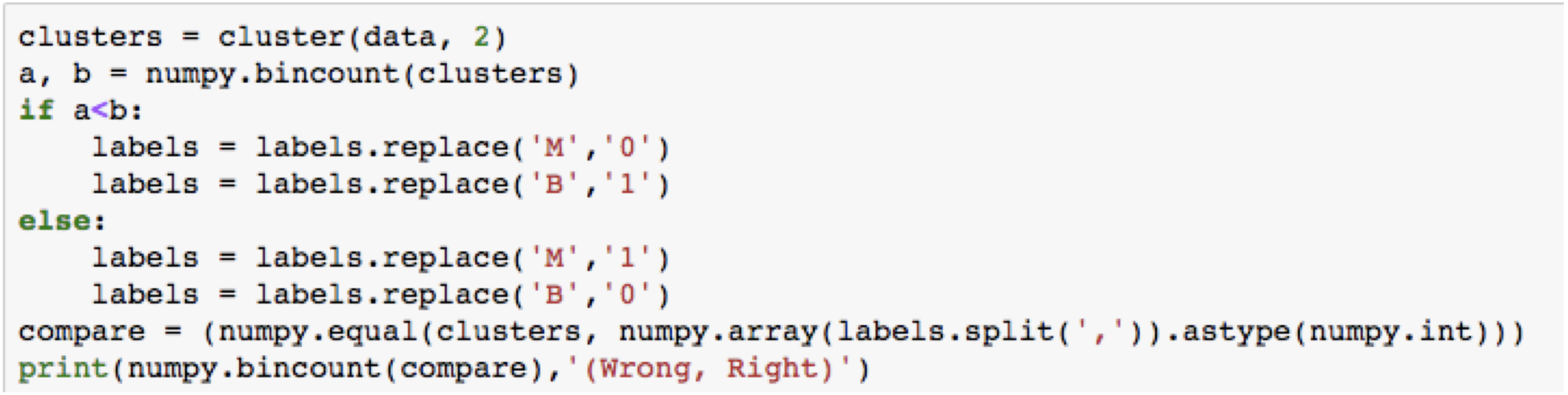
```python
import csv
import numpy

with open('wdbc.data') as input_file:
    text_data = [row for row in csv.reader(input_file, delimiter=',')]
for line in text_data:
    _ = line.pop(0) #We remove the ID - no need for it

known_labels = ','.join([line.pop(0) for line in text_data])
raw_data = numpy.array(text_data).astype(numpy.float)
data    = raw_data / numpy.max(raw_data, axis = 0)
```



```python
def all_dist(observation, data):
    return numpy.sqrt((data[:, 0] - observation[0])**2 + (data[:, 1] - observation[1])**2)

def cluster(data, k):
    samples, _= data.shape
    centroids = numpy.array([data[numpy.random.randint(samples), :,] for _ in range(k)])
    done = False
    while not done:
        distances = numpy.empty((k,samples))
        for d in range(k):
            distances[d, :] = all_dist(centroids[d], data)
        winners = numpy.argmin(distances, axis = 0)
        clusters = [data[winners == i, :] for i in range(k)]
        prev_centroids = centroids
        centroids = numpy.array([numpy.average(c, axis = 0) for c in clusters])
        if numpy.sum(prev_centroids-centroids) == 0:
            done=True
    return winners
```



```python
clusters = cluster(data, 2)
a, b = numpy.bincount(clusters)
if a<b:
    labels = labels.replace('M','0')
    labels = labels.replace('B','1')
else:
    labels = labels.replace('M','1')
    labels = labels.replace('B','0')
compare = (numpy.equal(clusters, numpy.array(labels.split(',')).astype(numpy.int)))
print(numpy.bincount(compare),'(Wrong, Right)')
```



```python
clusters = cluster(data, 2)
a, b = numpy.bincount(clusters)
if a<b:
    labels = labels.replace('M','0')
    labels = labels.replace('B','1')
else:
    labels = labels.replace('M','1')
    labels = labels.replace('B','0')
compare = (numpy.equal(clusters, numpy.array(labels.split(',')).astype(numpy.int)))
print(numpy.bincount(compare),'(Wrong, Right)')
```

```
[ 66 503] (Wrong, Right)
```

A common problem

We are extremely sensitive to the choice of initial random centroids

- We end up with finding local minimum when we really need a global minimum

The solution

Choose initial centroids non randomly

Chen Zhang and Shixiong Xia, " K-means Clustering Algorithm with Improved Initial center," in Second International Workshop on Knowledge Discovery and Data Mining (WKDD), pp. 790-792, 2009.

F. Yuan, Z. H. Meng, H. X. Zhangz, C. R. Dong, " A New Algorithm to Get the Initial Centroids," proceedings of the 3rd International Conference on Machine Learning and Cybernetics, pp. 26-29, August 2004.

Koheri Arai and Ali Ridho Barakbah, "Hierarchical K-means: an algorithm for Centroids initialization for k-means," department of information science and Electrical Engineering Politechnique in Surabaya, Faculty of Science and Engineering, Saga University, Vol. 36, No.1, 2007.

"My Solution"

Choose many more initial centroids

After clustering, fuse the two centroids that are closest to each other

Repeat until the required number of clusters is met

Cleaning

If there are outliers in the dataset they will be added to some cluster and may influence the precision

One may simply remove points that are far from centroids or from neighbors

IMPORTANT

In the daily use of KNN and K-means you don’t do it yourself

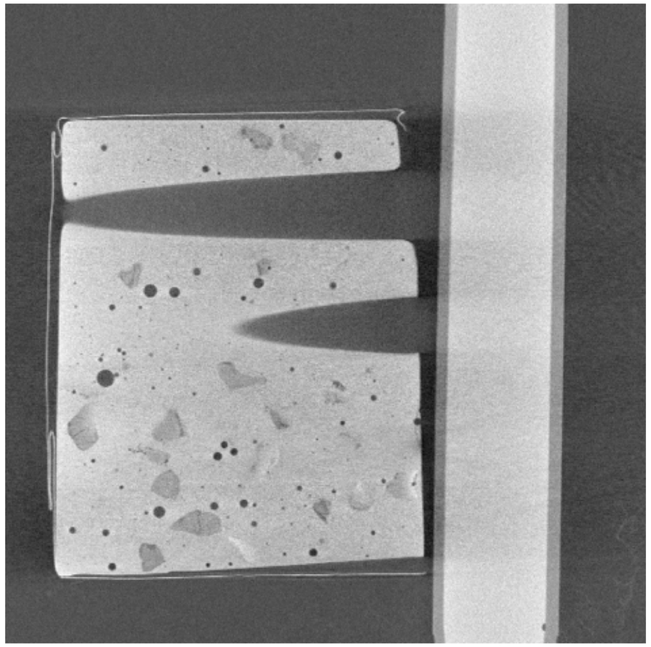
Use libraries

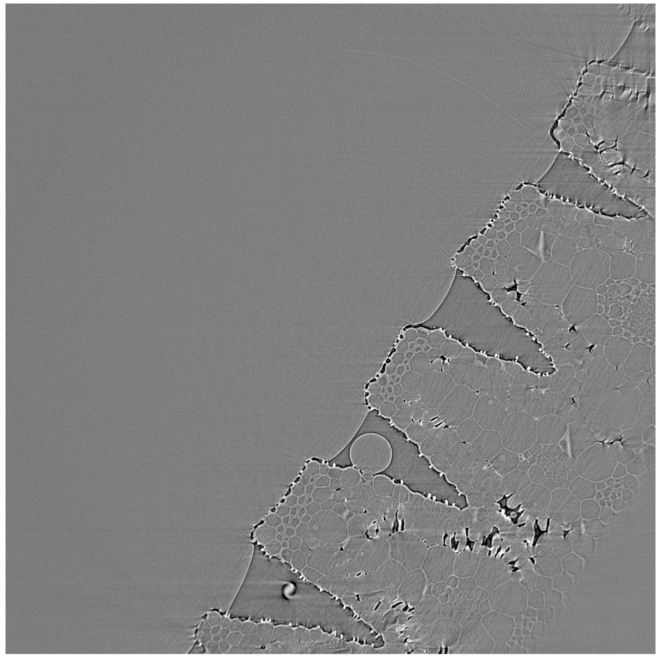
Scipy and scikit-learn

Quiz 3

Anybody in class have a problem where KMC may be used?

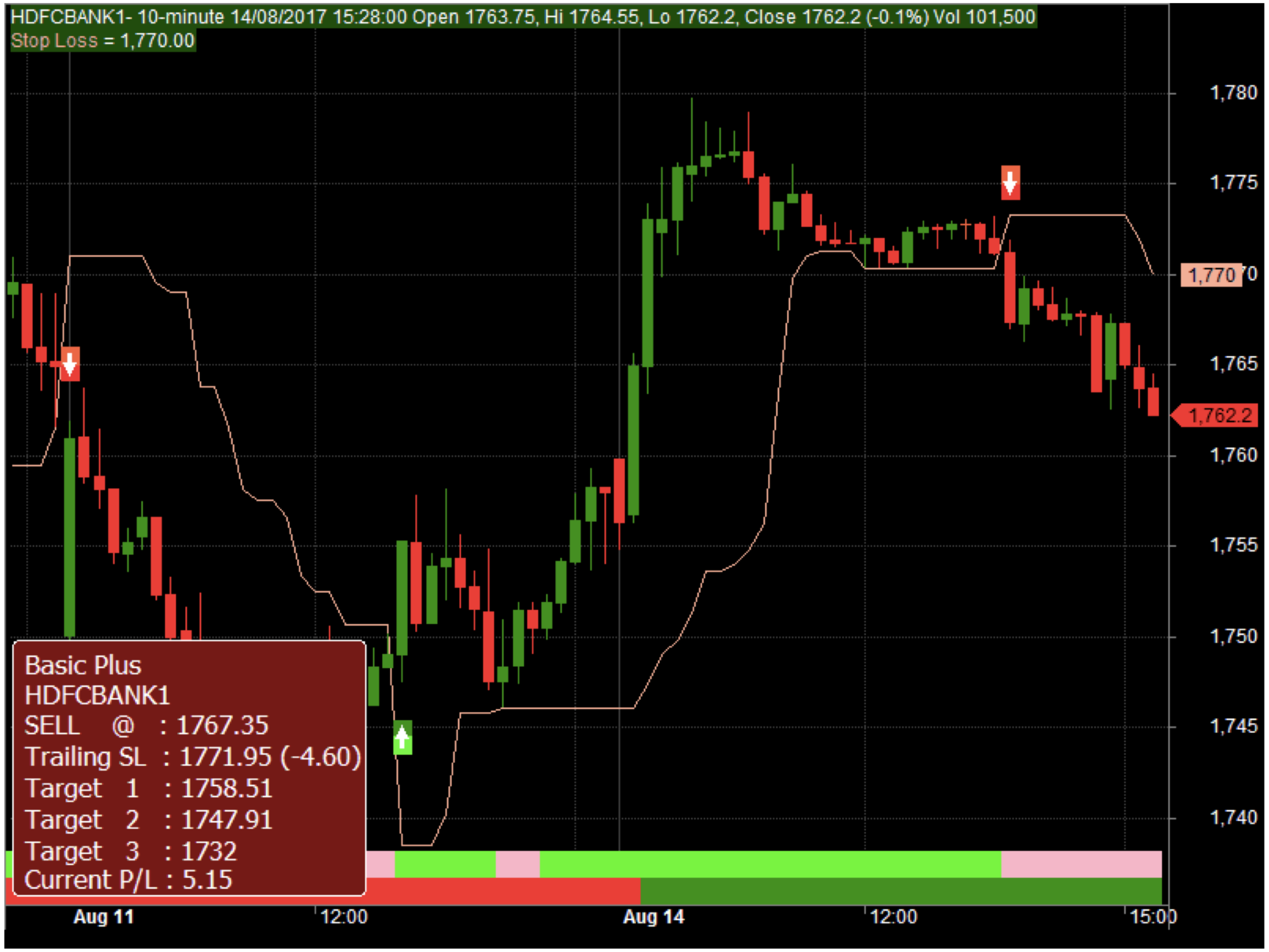
Use cases: Image segmentation





Use cases: Multi variate data

Use cases: Outlier detection



HDFCBANK1- 10-minute 14/08/2017 15:28:00 Open 1763.75, Hi 1764.55, Lo 1762.2, Close 1762.2 (-0.1%) Vol 101,500
Stop Loss = 1,770.00

Basic Plus
HDFCBANK1
SELL @ : 1767.35
Trailing SL : 1771.95 (-4.60)
Target 1 : 1758.51
Target 2 : 1747.91
Target 3 : 1732
Current P/L : 5.15

Possible dataset

- X-ray images of potatoes
  - Rotating while imaging
- Various noise levels
- Find the ones with holes or pins in
  - With as much noise present at possible