# K-NN FOR PORTFOLIO RECOMMENDATION

COMPUTATIONAL THINKING

MSIN0023 FINAL ASSIGNMENT

# I-System Requirements and the KNN algorithm

## a) The Business problem in Asset Management and relevancy of KNN

With the development of online banking, people have more opportunity to invest their money by themselves. On the other hand, online banking is also characterised by less bankers, able to provide quality wealth management, leaving people to invest on their own. However, most people being risk averse in terms of investment, professional advice is often a requirement when it comes to joining a bank. At the same time, AI and ML have made investing easier and cheaper as investors don't have to pay for stockbrokers anymore to sort out their stock transitions. Therefore, a challenge for online banks will be to provide an efficient investment recommendation system, easy to use for every type of investors.

In this assignment, I will demonstrate how my algorithm simulates a solution for online banks who would like to provide a wealth management tool to their clients. I will use collaborative filtering to create an asset bundle to a client. This asset bundle will consist of a selection of assets that the client does not have in their portfolio but is most likely to fit their risk profile.

Collaborative filtering is a predictive process thanks to which recommendation engines analyse the information about users with similar tastes to assess the probability that a target individual will enjoy a recommended product. For example, let Phil and Greg be two customers of our bank. If Phil and Greg have similar investing profiles and Greg buys an asset that Phil does not have, then the bank should recommend that Phil buys this asset.

KNN is one of the most common algorithms for collaborative filtering, which justifies this choice of algorithm for my business problem. KNN will simply find -for a given client- the K most similar (Nearest Neighbours) clients portfolio wise. Then, it will recommend assets based on the portfolios of the K nearest neighbours previously found.

## b) Functional and non-functional requirements of KNN

### a. Functional requirements of KNN

In order to understand the algorithm better, I will first describe the dataset I generated to simulate my business problem. The algorithm uses two data elements: the portfolio matrix (portfolioMatrixMaking) and the given client's asset information (portfoliostosort). More on asset information will be discussed in section IV.

Two major tasks have to be executed by the code with the inputted data. *(1)* First, using the portfoliostosort matrix (input), the algorithm will have to output the K nearest neighbours (K-NN) of the given client. *(2)* Then, the K-NNs found in task one (input), will have to be processed by the algorithm in order to return a defined number of recommended assets (final output).

There are sub-functional requirements to execute the K nearest neighbours' task *(1)*:
First the algorithm has to calculate and output the Euclidean distances between the given client and the rest of the clients of the bank (input) using this formula:

$$d(x,y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

The algorithm will then output a sorted list of those distances (input), allowing to find the K-NN (smallest distances).

We have sub-functional requirements to recommend the X bundle of assets *(2)*:
First the algorithm selects the K distances (input) to recommend the assets (output). The algorithm then finds and outputs the corresponding asset information thanks to the K distances as input. Finally, the algorithm inputs these assets information to sample and find the assets that were bought by the K neighbours but not by the given user (final output).

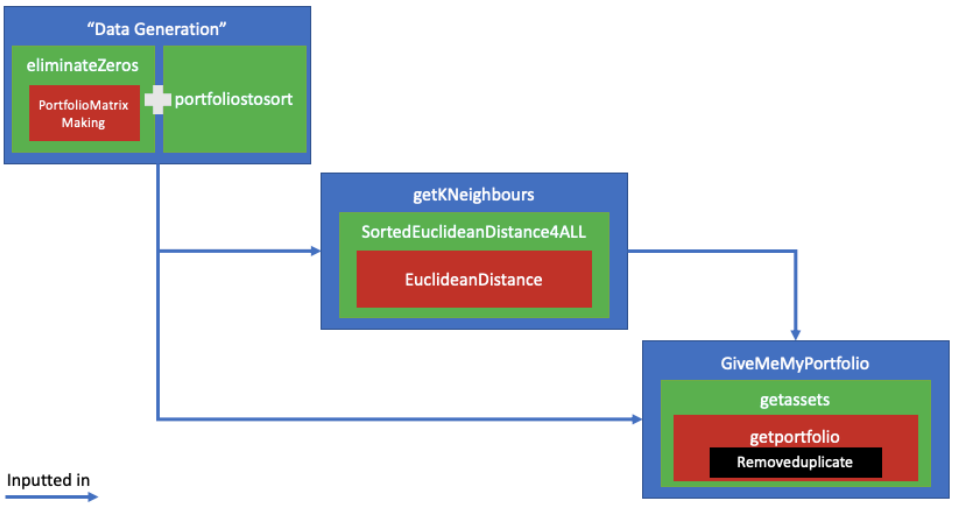
### b. Non-Functional requirements of KNN

The key non-functional requirement is performance with the data. As we focus on the banking industry, datasets can be enormous (Revolut has 7million users). Moreover, the number of assets is almost infinite. The algorithm's complexity of $O(n^2)$ that we will see in section III reflects the potential challenges of this algorithm. However, wealth management recommendations can be made monthly, which reduces considerably the complexity and allows our algorithm to meet the requirements under data load.
On the other hand, ease of use and quality are also key functional requirements. They are met as there are only 4 inputs in the algorithm which makes it simple to use. Also, the software only requires a clean dataset to run without bugs.

## II- Functioning of the code

### a) Construction of the code

The code is divided in two sections: one creates the input dataset, the other processes it. Section two has two sub-parts: one finds the K-NNs, the other explores the portfolios of the K-NNs of a given client to give him asset recommendations. The code is built on Andrew's template for K-NN I seminar 8.




### b) Flow of the code

#### a. Part 1: generating the input data

**The first part uses three functions to generate the data.**

| Function Name | Inputs | Outputs |
|---|---|---|
| portfolioMatrixMaking | Number of clients (portfolioqty) + max number of assets in the portfolio (assetqty) | List of lists of asset preferences per client |
| eliminateZeros | Matrix of clients + asset corresponding preferences | Matrix of clients without 0 |
| portfoliostosort | Number of clients' preferences (-1,0,1) to be analysed | Preferences of the given client |

In portfolioMatrixMaking, the sub lists are the clients' asset preferences which can be -1 (sold with loss), 0 (never bought), 1 (profitable). Those tastes are randomly generated by the function with the library random.
Then, eliminateZeros reduces the weight of the 0 values by replacing them with the column mean (-1 or 1).

Output given the input: (10,10):

```
In [9]: eliminateZeros(portfolioMatrixMaking(10,10))

Out[9]: [[1, 1, -1, -1, -1, 1, 1, -1, 1, 1],
         [-1, 1, -1, -1, -1, 1, 1, -1, 1, 1],
         [-1, 1, 1, -1, -1, 1, 1, -1, 1, 1],
         [1, -1, -1, -1, -1, 1, 1, -1, 1, 1],
         [-1, -1, -1, 1, -1, 1, 1, -1, 1, -1],
         [1, 1, -1, -1, 1, 1, 1, -1, -1, 1],
         [1, -1, -1, -1, -1, 1, 1, -1, -1, 1],
         [1, 1, 1, -1, -1, -1, 1, -1, 1, 1],
         [1, 1, -1, 1, 1, 1, -1, -1, 1, 1],
         [1, 1, 1, -1, -1, 1, -1, -1, -1, 1]]
```

Finally, portfoliostosort generates the characteristics (dataset) of the given user we will recommend assets to.

Output given the input (10):

```
In [10]: portfoliostosort(10)

Out[10]: [1, 0, -1, -1, 1, 1, -1, 0, 1, 0]
```

b. Part 2: processing the data of part 1

**The second part is divided in two parts: getting the K nearest neighbours and making the asset recommendations**

i. Getting the K nearest neighbours

| Function Name | Inputs | Outputs |
|---|---|---|
| EuclideanDistance | Data points | Distance between datapoints |
| SortedEuclideanDistance4ALL | Given client's preferences + all clients' matrix | List of sorted lists of Euclidean distances + corresponding client ID to the preferences |

| | | |
|---|---|---|
| getKNeighbour | Number of neighbours + client's preferences | K first user portfolios + distance from given user |

EuclideanDistance and SortedEuclideanDistance4ALL are the two pillars of the function getKNeighbour.

EuclideanDistance computes the Euclidean distance between two data points.
Then, built on EuclideanDistance, SortedEuclideanDistance4ALL first computes the distances between the given client's preference vector and the preference vector of each of the other clients. After computing the distances, SortedEuclideanDistance4ALL sorts them in ascending order
Finally, getKNeighbour selects K lists of preferences in the list generated by SortedEuclideanDistance4ALL. These lists are the first preferences of the output of SortedEuclideanDistance4ALL and the K-NNs of the given client.

Testing getKNeighbour:

```
In [13]: PMM = eliminateZeros(portfolioMatrixMaking(1000,1000))
         PTS = portfoliostosort(1000)

         getKNeighbour(PMM,PTS,5)

Out[13]: [[369, 38.8329756778952],
          [110, 38.98717737923585],
          [167, 39.03844259188627],
          [571, 39.03844259188627],
          [230, 39.089640571384131]]
```

ii. Making recommendations

| **Function name** | **Inputs** | **Outputs** |
|---|---|---|
| RemoveDuplicate | Portfolio matrix | Unique Portfolio matrix |
| getportfolio | Output of getKNeighbour | Vector of assets from K neighbours |
| getassets | K asset characteristics | Profitable asset ID in K portfolios |
| GiveMeMyPortfolio | K profitable assets ID | Sample of recommended asset ID |

RemoveDuplicate, getportfolio and getassets are the three pillars of GiveMeMyPortfolio, which gives the final recommendations to the user.
First, RemoveDuplicate removes any similar ID for accuracy improvements.
Then, the function getportfolio takes the output of the function getKNeighbour: client IDs, and their associated asset information vector stored in the clients' matrix.
Subsequently, getassets uses the K asset characteristics obtained previously and stores each asset ID that is profitable to one of the K clients but never bought by the given client.
Therefore, we have a list of assets profitable to the K neighbours but that the given client never bought. The function GiveMeMyPortfolio can then create a sample of X asset ID which can be recommended to our given client.

Output of GiveMeMyPortfolio:

```
In [15]: PMM = eliminateZeros(portfolioMatrixMaking(1000,1000))
         PTS = portfoliostosort(1000)

         GiveMeMyPortfolio(PMM,PTS,5,10)

Out[15]: [243, 495, 51, 954, 477, 361, 622, 579, 942, 800]
```

### c) Generalisation of the code and error management

getKNeighbour and GiveMeMyPortfolio are the two main functions of the code, therefore, they are generalised. getKNeighbour can be used in any problem solved by a KNN algorithm while GiveMeMyPortfolio can be used in any item based collaborative filtering.
Error Management is only handled in GiveMeMyPortfolio, mainly by eliminating data-based errors. The data goes through 3 filters:

- Is the portfolio matrix a list of lists and the given client's matrix a list?
- Are all asset lists the same length?
- Is the size of the sample of recommendation smaller or equal to the total of assets available?

If one or more of these questions' answer is no, an appropriate error message is displayed, and the code stops to run.

## III- code complexity analysis

First, I created different data sizes to run them through my code in order to assess its complexity. The data is an n*n portfolio matrix with n ∈ {100,200,300,400,500,600,700, 800, 900,1000} items in the asset characteristics vector.

Code for this step:

```
In [16]: #We generate a fixed dataset to analyse the loops and running time of the code

         nvals = []

         for i in range (100, 1001, 100): #creation of a list of different n values
             nvals.append(i)

         print(nvals)

         dif_PMM = []
         dif_PTS = []

         for value in nvals: #The different n values are inputed in the function generating the data.
                             #The different data sizes are stored in the list
             dif_PMM.append(eliminateZeros(portfolioMatrixMaking(value,value)))
             dif_PTS.append(portfoliostosort(value))


[100, 200, 300, 400, 500, 600, 700, 800, 900, 1000]
```

Once I had the different data sizes, I created a Big O plot for time performance analysis.

Code for the plot:

In [19]:
```
#creation of a Big(o) plot for running time

plt.style.use('ggplot')
plt.rcParams['figure.figsize'] = [15, 10]

timeval = []
for i in range (len(dif_PMM)):
    start_time = time.time()
    GiveMeMyPortfolio(dif_PMM[i],dif_PTS[i],5,5)
    timeval.append(time.time()-start_time)

plt.xlabel('size of matrix n*n and preference vector n')
plt.ylabel('time taken in seconds')
plt.title('time taken by the recomendation engine for different data sizes')
plt.plot(nvals, timeval, 'o-');
```

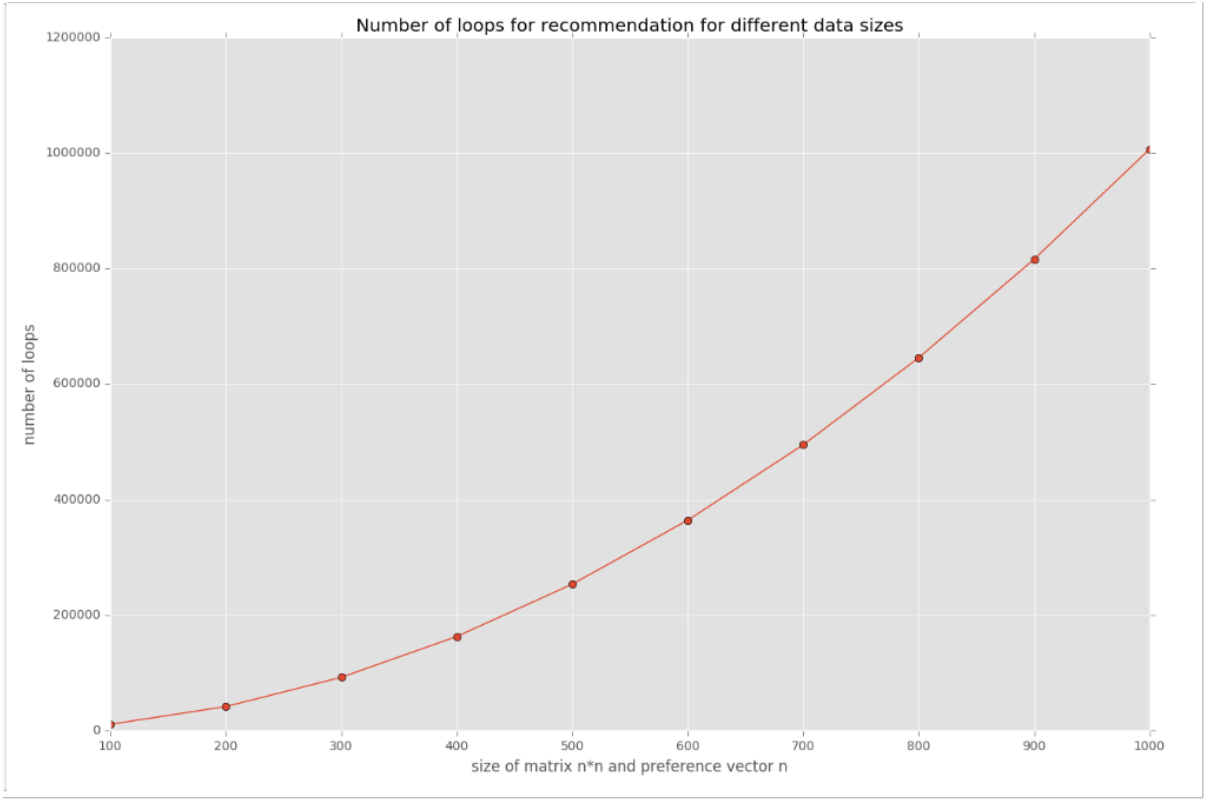

The graph displays a polynomial shaped line. However, we can't rely only on running time as it also correlated with the available computational power during the test. Thus, I also did a Big O plot using a loop counter. There, it shows a complexity of $O(n^2)$ is displayed in the graph below.

In [21]:
```
#creation of a Big(o) plot for loopcounting

#Style plot
plt.style.use('ggplot')
plt.rcParams['figure.figsize'] = [15, 10]

loopvals = []
for i in range (len(dif_PMM)):
    loopcounter = 0
    GiveMeMyPortfolio(dif_PMM[i],dif_PTS[i],5,5)
    loopvals.append(loopcounter)

plt.xlabel('size of matrix n*n and preference vector n')
plt.ylabel('number of loops')
plt.title('Number of loops for recommendation for different data sizes')
plt.plot(nvals, loopvals, 'o-');
```




With such complexity, the algorithm is fallible. For instance, if we ran the code Revolut's (7million clients) or HSBC's (40million clients) benefit, the matrices would be enormous (X possible assets * 40'000'000). Moreover, the algorithm would have to run for all of the clients, not just one like here. The running time and the loops being exponential, we would have to find more suitable adaptations.

## IV- Test data description

The data was generated with portfolioMatrixMaking and portfoliostosort functions described in section II. There are two datasets: the portfolios matrix with all of the clients' portfolios' information and the given user's asset characteristics. I inputted a variation of n and m to change the matrix size and asset information (n) to test my code (more in part III).

The portfolio matrix portfolioMatrixMaking has m rows representing the portfolio of each clients. Each item in a row represents a client's asset information vector. The n columns represent all of the clients' asset information for a particular asset. Each client has an ID number: ID 10 are the asset information of the client's portfolio located on the 11$^{th}$ row of the portfolio matrix. Similarly, each asset also has a related number: ID 10 then is 11$^{th}$ column of the matrix. The matrix is constructed as a list of lists in python without using NumPy library as all of my matrix operations were realisable without NumPy. Each item in the sub lists is an asset information (-1,0,1). Instead of using NumPy, I used the library random to randomly generate each information.
The given client matrix portfoliostosort give this special client's portfolio details in a python list to allow us to later make recommendations.

I chose to create my own dataset to simplify code testing with different data sizes and content.

## V- Conclusion

The code follows the principle of simplicity as it uses function and variable names that are understandable and related to the problem (e.g.: portfolioMatrixMaking). Moreover, each function being essential to run the code, extra loops are avoided, and the code is minimalised and well designed.
code follows the principle of aligned functionality as all function depend on the previous one, which creates a clear hierarchy.
Additionally, as there are only 4 attributes to plug in the final code to make the algorithm work, the code is easy to use. Additionally, for the previous reason but also because the wanted output is directly given (asset bundle), the code understands the users' needs perfectly.
With these three requirements met, the code can be characterised as well designed.

If the complexity of the code is here not a problem, the code is limited, especially if it is in a cold start situation. In other words, it cannot be used when launching an online platform/bank as there will not be enough users to be compared and results on the assets can take several months to appear. here the running capacity of the code will not be threatened, only its accuracy.

Word count: 1996

# Code Appendix

## Main code

### KNN Portfolio

**Load Libraries**

In [6]:
```
#load useful libraries
import operator
import random
import time
import matplotlib
from collections import Counter
from statistics import *

#Import matplotlib
%matplotlib inline
from matplotlib import pyplot as plt
```

**1. I will start by creating a simulation of a dataset that could be used in a portfolio management business problem.**

In [*]:
```
def portfolioMatrixMaking(portfolioqty,assetqty):

    '''
    Creation of a portfolio matrix with a list of lists. portfolioqty gives the
    quantity of portfolios that will be in the matrix. The number of
    assets in each portfolios will be defined by assetqty.

    '''

    items = [-1, 0, 1]
    portfolioMatrix = []

    for i in range(portfolioqty):
        profile_portfolios = []

        for j in range(assetqty):
            profile_portfolios.append(random.choice(items))

        portfolioMatrix.append(profile_portfolios)

    return portfolioMatrix

def column(matrix,j):
    '''
    Here we select a row j of the portfolio matrix.

    '''
    return [row[j] for row in matrix]
```

```
def eliminateZeros (matrix):
    '''
    This function replaces 0 values with the column mean (-1 or 1) in order to reduce the
    weight of zero values on the euclidean distance

    '''

    for j in range (len(matrix[0])):
        columnMean = mean(n for n in column(matrix,j)if n!=0)

        for i in range(len(matrix)): #if the mean of the column<0 then each 0 is replaced by -1
                                     #otherwise they are replaced by 1

            if matrix[i][j] == 0 and columnMean < 0:
                matrix [i][j] = -1
            elif matrix [i][j] == 0 and columnMean >= 0:
                matrix[i][j] = 1

    return matrix

def portfoliostosort(portfolioqty):

    '''
    This function generates a unique portfolio vector.
    This vector represents the person(portfolio) we need to recommend
    the portfolio list to. The lenght of this vector is
    equal to the input: "portfolioqty".

    '''

    items = [-1, 0, 1]
    profile_portfoliosbis = []

    for i in range(portfolioqty):
        profile_portfoliosbis.append(random.choice(items))

    return profile_portfoliosbis
```

**2. I will then initialize the functions that will help me solve subproblems of the K-NN algorithm.**

In [9]:
```
loopcounter = 0

def EuclideanDistance(A,B):

    '''
    The aim of this function is to
    calculate the euclidean distance between
    two given datapoints A and B.

    '''
    global loopcounter

    distance = 0
    for i in range (len(A)):
        loopcounter += 1

        distance += (A[i]-B[i])**2
    euclideandistance = distance**0.5

    return euclideandistance
```

```
def SortedEuclideanDistance4ALL(data,newportfolio):
    '''
    Input: Portfolio matrix and the given assets for each portfolio
    This function computes the Euclidean distance between
    the given portfolio and all of the rest of the portfolios in the
    portfoliomatrix. Then the function sorts this distance in ascending order.
    Output: list of lists of the form [[portfolio_ID, distance],...].

    '''
    global loopcounter

    liste_distance = []
    for i in range(len(data)):
        loopcounter += 1

        liste_distance.append([i,EuclideanDistance(data[i],newportfolio)])
            #It is the Euclidean distance between the given portfolio and the rest of the portfolios.

    sorted_edistance = sorted(liste_distance,key=operator.itemgetter(1))
    #It is the list of portfolios in ascending euclidean distances.

    return sorted_edistance
```

**3. The next function will call on the K Nearest Neighbours.**

In [12]:
```
def getKNeighbour(data,newportfolio,k):

    '''
    This function gets the K nearest neighbour of
    our given portfolio. It uses the portfolio matrix, the given portfolio's assets
    and the number of K neighbours to be used.

    '''

    global loopcounter
    sorted_edistance = SortedEuclideanDistance4ALL(data,newportfolio)
    #Here we the output of the SortedEuclideanDistance4ALL function
    Kneighbour = []

    for i in range(k):
        loopcounter += 1

        Kneighbour.append(sorted_edistance[i])
        #The K first lists of the sorted list of Euclidean distances are kept

    return Kneighbour
```

**3.1 Test of our getKNeighbour function.**

In [13]:
```
PMM = eliminateZeros(portfolioMatrixMaking(1000,1000))
PTS = portfoliostosort(1000)

getKNeighbour(PMM,PTS,5)
```

Out[13]:
```
[[369, 38.8329756778952],
 [110, 38.98717737923585],
 [167, 39.03844259188627],
 [571, 39.03844259188627],
 [230, 39.08964057138413]]
```

**4. These next functions will allow us to analyse our data accurately**

In [*]:
```
loopcounter = 0

def RemoveDuplicate(listing):

    '''This function removes any similar portfolios in order to have a more accurate analysis.'''

    global loopcounter

    Notanyduplicate = []
    for i in range(len(listing)):
        loopcounter += 1
        if listing[i] not in Notanyduplicate: #Elements are added in the list iff they are not already in it.
            Notanyduplicate.append(listing[i])

    return Notanyduplicate

def getportfolio(data,newportfolio,k):

    '''
    This function gets the K rows of the matrix thanks to the K porfolio ID.
    The IDs come from the output of the function getKneighbour.

    '''
    global loopcounter

    asset_vector = []
    list_neighbours = getKNeighbour(data,newportfolio,k) #Use of the output of the getKNeighbour function

    for i in range(len(list_neighbours)):
        loopcounter += 1

        asset_vector.append(data[list_neighbours[i][0]])
        #The loop gets the first element of the lists [portfolio_ID, distance].
        #Thus, data[portfolio_ID] --> the correct row is obtained and stored.
    return asset_vector
```

```
def getassets(data,newportfolio,k,assetqty):

    '''
    This function takes the output of the fuction getportfolio and a particular asset.
    each asset detained by one of the K neighbours and not by the particular portfolio
    is stored in the list. The function RemoveDuplicate removes any similar ID.

    '''

    global loopcounter

    asset_vector = getportfolio(data,newportfolio,k)
    list_assets = []

    for i in range(len(asset_vector)):
        for j in range(len(asset_vector[0])): #The loops go through each values of the preference of the K neighbours.
            loopcounter += 1

            if asset_vector[i][j] == 1 and newportfolio[j] == 0:
                list_assets.append(j) #The number j represents he number of the matrix column corresponding
                                      #to the portfolio ID.

    return RemoveDuplicate(list_assets)
```

**5. I will now start the Recommendations Engine**

In [15]:
```
def GiveMeMyPortfolio (data, newportfolio, k, assetqty):

    '''
    This function tests if the inputted data is clean. Then it creates a sample of
    X assets which are the assets to be recommended

    '''

    global loopcounter
#First we check if the data can be used. If not, an error message is displayed.
    if type(data) != list or type(data[0])!= list or type(newportfolio) != list:
        raise Exception("invalid data type: the data you entered is not a list of list")

    elif len(data[0]) != len(newportfolio):
        raise Exception("the preference vectors are not similar")

    elif assetqty > len(newportfolio):
        raise Exception("the number of asset you require is bigger than the number of assets available")

#Once we know the data is clean, we can process it.
    else:
#The two if statements change the size of K and of the asset list if the number of assets available smaller than them.
        if k > len(data):
            k = len(data)
        assets = getassets(data, newportfolio, k, assetqty) #Getting the output from the previous function.

        if assetqty > len(assets):
            assetqty = len(assets)

    return random.sample(assets,assetqty)

#We then make a function that creates a sample of assets for the portfolio
```

In [16]:
```
PMM = eliminateZeros(portfolioMatrixMaking(1000,1000))
PTS = portfoliostosort(1000)

GiveMeMyPortfolio(PMM,PTS,5,10)
```

Out[16]:
```
[118, 539, 458, 244, 704, 948, 74, 831, 648, 254]
```

# Analysis code

**6. behaviour of the model with different datasizes**

In [18]:
```
#We generate a fixed dataset to analyse the loops and running time of the code

nvals = []

for i in range (100, 1001, 100): #creation of a list of different n values
    nvals.append(i)

print(nvals)

dif_PMM = []
dif_PTS = []

for value in nvals: #The different n values are inputed in the function generating the data.
                    #The different data sizes are stored in the list
    dif_PMM.append(eliminateZeros(portfolioMatrixMaking(value,value)))
    dif_PTS.append(portfoliostosort(value))
```

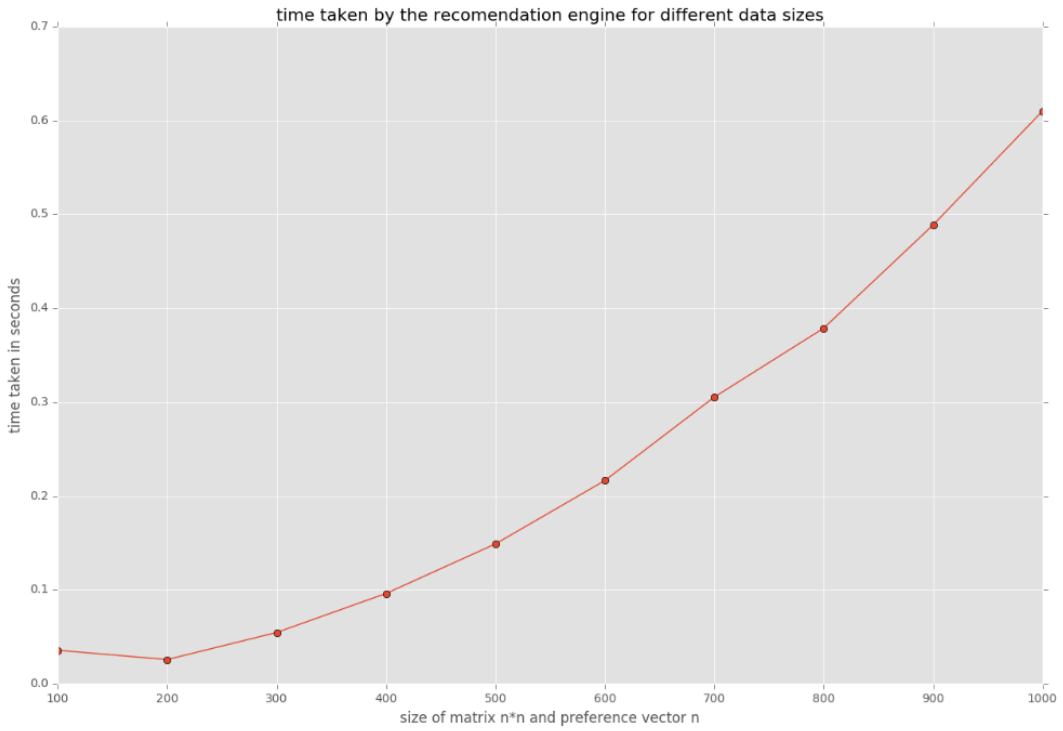
```
[100, 200, 300, 400, 500, 600, 700, 800, 900, 1000]
```

In [19]:
```
#creation of a Big(o) plot for running time

plt.style.use('ggplot')
plt.rcParams['figure.figsize'] = [15, 10]

timeval = []
for i in range (len(dif_PMM)):
    start_time = time.time()
    GiveMeMyPortfolio(dif_PMM[i],dif_PTS[i],5,5)
    timeval.append(time.time()-start_time)

plt.xlabel('size of matrix n*n and preference vector n')
plt.ylabel('time taken in seconds')
plt.title('time taken by the recomendation engine for different data sizes')
plt.plot(nvals, timeval, 'o-');
```

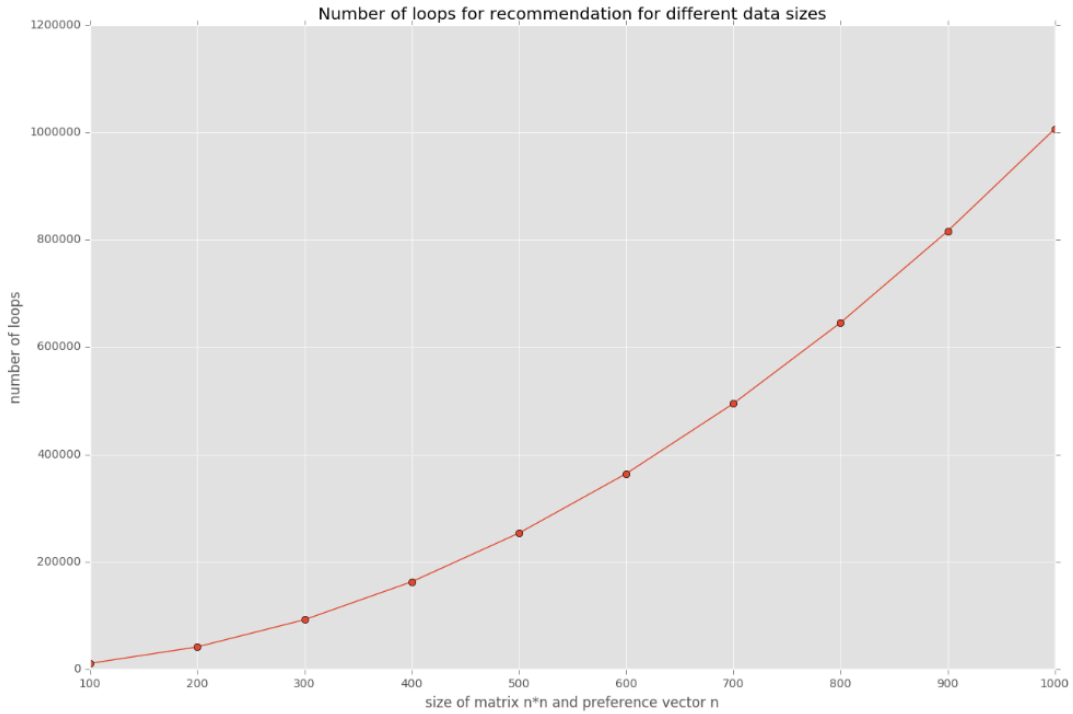


```
In [21]: #creation of a Big(o) plot for loopcounting

         #Style plot
         plt.style.use('ggplot')
         plt.rcParams['figure.figsize'] = [15, 10]

         loopvals = []
         for i in range (len(dif_PMM)):
             loopcounter = 0
             GiveMeMyPortfolio(dif_PMM[i],dif_PTS[i],5,5)
             loopvals.append(loopcounter)

         plt.xlabel('size of matrix n*n and preference vector n')
         plt.ylabel('number of loops')
         plt.title('Number of loops for recommendation for different data sizes')
         plt.plot(nvals, loopvals, 'o-');
```




## Data visualisation code

**7. Data visualisation**

```
In [9]: eliminateZeros(portfolioMatrixMaking(10,10))

Out[9]: [[1, 1, -1, -1, -1, 1, 1, -1, 1, 1],
         [-1, 1, -1, -1, -1, 1, 1, -1, 1, 1],
         [-1, 1, 1, -1, -1, 1, 1, -1, 1, 1],
         [1, -1, -1, -1, -1, 1, 1, -1, 1, 1],
         [-1, -1, -1, 1, -1, 1, 1, -1, 1, -1],
         [1, 1, -1, -1, 1, 1, 1, -1, -1, 1],
         [1, -1, -1, -1, -1, 1, 1, -1, -1, 1],
         [1, 1, 1, -1, -1, -1, 1, -1, 1, 1],
         [1, 1, -1, 1, 1, 1, -1, -1, 1, 1],
         [1, 1, 1, -1, -1, 1, -1, -1, -1, 1]]

In [10]: portfoliostosort(10)

Out[10]: [1, 0, -1, -1, 1, 1, -1, 0, 1, 0]
```