Faculty of Electronics and Information Technology
Warsaw University of Technology

Data Mining

Implementation K-medoids (PAM) algorithm for clustering objects with nominal and numerical attributes

Jakub Wieczorek

Szczecin, 2021

Contents

1. Description of the task

The k-medoids problem is a clustering problem similar to k-means. Both the k-means and k-medoids algorithms are partitional (breaking the dataset up into groups) and attempt to minimize the distance between points labeled to be in a cluster and a point designated as the center of that cluster. In contrast to the k-means algorithm, k-medoids chooses actual data points as centers, and thereby allows for greater interpretability of the cluster centers than in k-means, where the center of a cluster is not necessarily one of the input data points (it is the average between the points in the cluster). Furthermore, k-medoids can be used with arbitrary dissimilarity measures, whereas k-means generally requires Euclidean distance for efficient solutions. Because k-medoids minimizes a sum of pairwise dissimilarities instead of a sum of squared Euclidean distances, it is more robust to noise and outliers than k-means.

k-medoids is a classical partitioning technique of clustering that splits the data set of n objects into k clusters, where the number k of clusters assumed known a priori (which implies that the programmer must specify k before the execution of a k-medoids algorithm).

The medoid of a cluster is defined as the object in the cluster whose average dissimilarity to all the objects in the cluster is minimal, that is, it is a most centrally located point in the cluster.

1.1. Algorithm – Partitioning Around Medoids (PAM)

PAM uses a greedy search which may not find the optimum solution, but it is faster than exhaustive search. It works as follows [1]:

1. (BUILD) Initialise: greedily select k of the n data points as the medoids to minimise the cost
2. Associate each data point to the closest medoid.
3. (SWAP) While the cost of the configuration decreases:
   a) For each medoid m, and for each non-medoid data point o:
      i. Consider the swap of m and o, and compute the cost change
      ii. If the cost change is the current best, remember this m and o combination
   b) Perform the best swap of m_{best} and o_{best}, if it decreases the cost function. Otherwise, the algorithm terminates

Project is implemented in Python version 3,8 as a script with the help of matplotlib and numpy.

2. Description of the form of input and output data

Python script takes as parameters csv file path with a set of data. Each column represents a feature for example weight or price or values not related to any physical characteristic whereas each row the particular values. Result (output) of the algorithm is displayed on the plot – clusters items belonging to the cluster are characterised by the same colour. In addition medoids are marked. Moreover, output json file contains the full and neat result of classification.

Script arguments are filtered and validated using getopt library, so two forms of parameters, short and long are possible. Amount of clusters has to be specified in advance, by the parameter

-c or clusters. If parameters are wrongly specified, the help is printed. As it was stated already input data are stored in the csv. Parameter -d or delimiter specifies the data delimiter for example semicolon.

3. Description how to use a software

Software is a script which may be launched from the command line:

```
python3 KMedoids.py -i data.csv -c 2 --delimiter=";"
```

4. Description of all important design

Figure 4.1 shows the high-level logic chain. At the beginning input parameters are validated and parsed. In the result csv file name is extracted as well as csv file delimiter and amount of clusters. Subsequently csv file is parsed to the data structure. KMedoids object is created and its main API method fit is called which fits the data into clusters and chooses the medoids. Eventually the result is presented on the plot.

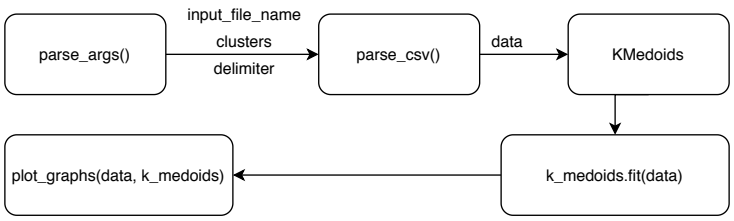

Figure 4.1. Functions call diagram

4.1. Arguments parsing

Following two listings show the script arguments parsing. Getopt functionality was exploited for this purpose. Script may be launched using short and long parameters – for example -i my_csv_file.csv or –input=my_csv_file.csv. Validation prevents from running the script with different arguments than defined in the help.

```
1  def parse_args(argv):
2      input_file_name = None
3      clusters        = None
4      delimiter       = None
5
6      if len(argv) < 3:
7          print('python KMedoids.py -i <input_file> -c <cluster_amount>')
8          sys.exit(2)
9
```

```
    try:
        opts, args = getopt.getopt(argv, "hi:c:d:", ["help", "input=",
            "clusters=", "delimiter="])
    except getopt.GetoptError:
        print('python KMedoids.py -i <input_file> -c <cluster_amount>')
        sys.exit(2)

    for opt, arg in opts:
        if opt in ('-i', '--input'):
            input_file_name = arg
        elif opt in ('-c', '--clusters'):
            clusters = int(arg)
        elif opt in ('-d', '--delimiter'):
            delimiter = arg
        elif opt in ('-h', '--help'):
            show_help()
            sys.exit(2)

    return input_file_name, clusters, delimiter
```

Consequently the csv parsing into intrinsic data list structure:

```
def parse_csv(file_name, delimiter=";"):
    with open(file_name, "r") as csv_file:
        content = csv.reader(csv_file, delimiter=delimiter)

        data = []
        # avoid first column and row (descriptive cells)
        column_amount = len(next(content))
        dict = {} # for json mapping
        row_number = 0
        for row in content:
            dict[row_number] = row[0]
            row_number += 1
            data.append([float(i) for i in row[1:column_amount]])

        return data, dict
```

4.2. PAM algorithm

Main function of KMedoids class is fit, which assigns data to clusters and chooses medoids. Figure 4.2 shows main KMedoids methods invocation order. At the beginning KMedoids object is created, then API fit function is called, which internally calls __initialize_medoids, __calculate_clusters and __update_clusters.

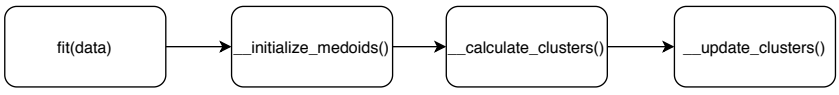

Figure 4.2. PAM algorithm main functions call diagram

4.2.1. Initialise

Initialise part greedily selects k of the n data points as the medoids to minimise the cost (distance). First medoid is random. Next ones are chosen basing on the maximum distance from this medoid to the rest of the data.

```
self.medoids.append(random.randint(0, self.__rows - 1))
while len(self.medoids) != self.n_cluster:
    self.medoids.append(self.__find_distant_medoid())
```

Method __find_distant_medoid for every row:

1. Calculate the shortest distance from the set of all medoids
2. Sort the distances from min to max
3. Filter them to take into account only those from <start_prob_dist*len(distances_index); end_prob_dist(distances_index)>, for example for 20 rows, start_prob_dist == 0.8 and end_prob_dist == 0.8 -> <16; 18>
4. Randomly choose from the above range one distance
5. Return the index row, so the data row for this distance

```
def __find_distant_medoid(self):
    distances = []  # contains the distances of each row entry from the medoid
    indices = []  # rows == indices, so 0, 1, 2, 3, etc.
    for row in range(self.__rows):
        indices.append(row)
        nearest_medoid, nearest_distance = self.
            __calculate_shortest_distance_to_medoid(row, self.medoids)
        distances.append(nearest_distance)
    distances_index = np.argsort(distances)  # distance indexes from min to max
    chosen_dist = self.__select_distant_medoid(distances_index)
    return indices[chosen_dist]
```

Method __select_distant_medoid takes one of the distant medoids index: start_index bases on the start_prob_dist, so for example start_prob_dist == 0.8, end_prob_dist == 0.9, there are 20 distant indexes (sorted from min to max!, the further the more distant), then start_index == round(0.8*20) == 16, so it will take from 16th to the 18th.

4.2.2. Calculate clusters

This part assigns data to the clusters basing on the minimum distance.

1. Associate each data point to the closest medoid -> define clusters
2. For each cluster calculate the distance from its medoid to all the cluster members
3. Scale the distance to the amount of cluster members

```
# define clusters and calculate distances
for row in range(self.__rows):
    nearest_medoid, nearest_distance = self._
        _calculate_shortest_distance_to_medoid(
            row, medoid_indexes)
    cluster_distances[nearest_medoid] += nearest_distance
    clusters[nearest_medoid].append(row)

# scale the distances
for medoid_idx in medoid_indexes:
```

```
    cluster_distances[medoid_idx] /= len(clusters[medoid_idx])
return clusters, cluster_distances
```

4.2.3. Update clusters

This is the most challenging SWAP part.

```
for i in range(self.max_iter):
    cluster_dist_with_new_medoids = self.__swap_and_recalculate_clusters()
    if self.__cost_decreases(cluster_dist_with_new_medoids):
        self.clusters, self.cluster_distances = self.__calculate_clusters(self.medoids)
    else:
        break
```

The following listing shows the implementation with the necessary explanation comments.

```
def __swap_and_recalculate_clusters(self):
    cluster_dist = {}
    # For each medoid m
    for m in self.medoids:
        cost_changed = False

        # and for each non-medoid data point o
        for o in self.clusters[m]:
            if o != m:
                # swap
                cluster_list = list(self.clusters[m])
                cluster_list[self.clusters[m].index(o)] = m
                cluster_list[self.clusters[m].index(m)] = o

                # compute the cost change
                new_distance = self.calculate_cluster_distance(o, cluster_list)

                # If the cost change is the current best, remember
                # this m and o combination
                if new_distance < self.cluster_distances[m]:
                    cluster_dist[o] = new_distance
                    cost_changed = True
                    break

        # If the cost change is the current best
        # remember this m and o combination
        if not cost_changed:
            cluster_dist[m] = self.cluster_distances[m]

    return cluster_dist
```

5. Experiments

Several set of experiments were prepared. Dataset contains models of cars of a few brands like Nissan NV200, Nissan Micra or BMW X1 and theirs maximal speed taken from the official sources [1] or acceleration time. Algorithm classifies them into 2 or 3 clusters. There are 20 cars at maximum to classify. Secondly a dataset of fruits[2] is tested. Data contain vitamins and other minerals saturation for 22 fruits. Results are saved in the json format as well as displayed on the figures. Cars dataset is presented on the Table 5.1 and fruits on Table 5.2.

Table 5.1. Cars dataset

	Cena [PLN]	Prędkość maksymalna [km/h]	Przyspieszenie do 100 [s]
Nissan NV200	50890	165	15
Nissan Micra	61900	178	11.8
Nissan Juke	76930	180	10.7
Nissan Qashqai	98480	198	9.9
Nissan X-Trail	115440	198	11.5
Nissan Leaf	123900	157	7.3
Nissan Navara	147000	184	10.8
Nissan GT-R	527000	315	2.9
BMW Seria 1	106400	250	4.8
BMW Seria 2	113700	250	4.6
BMW X1	133900	235	6.5
BMW X2	139100	250	5
BMW Seria 4	172900	250	4.5
BMW Seria 5	197900	305	3.4
BMW Seria 6	260900	250	5.3
Dacia Dokker	41550	173	12.5
Dacia Duster	42900	200	10.4
Dacia Lodgy	61800	185	10.9
Dacia Sandero	32900	182	11.5

[1] `https://www.autocentrum.pl/nowe/nissan/`
[2] `http://www.kups.org.pl/files/?id_plik=1378`

Table 5.2. Fruits dataset in 100g flesh

	K(mg)	Cu(mg)	Mn(mg)	Wit.C(mg)	B1(mg)	Folate(µg)	b-kar(µg)	A	K(µg)
Gooseberry	198	0.07	144	27.7	0.04	6	0	15	0
Black lilac	280	61	0	36.0	0.07	6	0	30	0
Blueberry	77	57	336	9.7	37	6	32	3	19.3
Peach	190	68	61	6.6	24	4	162	16	2.6
Sweet Cherry	222	60	70	7.0	27	4	38	3	2.1
Pear	119	82	49	4.2	12	7	13	1	4.5
Apple	107	27	35	4.6	17	3	27	3	2.2
Blackberry	162	165	646	21.0	0.02	25	128	11	19.8
Raspberry	151	0.09	670	26.2	32	21	12	2	7.8
Apricot	259	78	77	10	0.03	9	1094	96	3.3
Black currant	322	86	256	181.0	0.05	0	0	12	0
Red currant	275	107	186	41.0	0.04	8	25	2	11
Plums	157	57	52	9.5	28	5	190	17	6.4
Strawberry	153	48	386	58.8	24	24	7	1	2.2
Grape	191	127	71	3.2	69	2	39	3	14.6
Cherry	173	104	112	10.0	0.03	8	770	64	2.1
Cranberry	85	61	360	13.3	12	1	36	3	5.1
Oranges	181	45	25	53.2	87	30	71	11	0
Grapefruit	139	47	12	34.4	36	10	552	46	0
Tangerine	166	42	39	26.7	58	16	155	34	0
Hazelnut	680	1725	6175	6.3	643	113	11	1	14.2
Walnut	441	1586	3414	1.3	349	98	12	1	2.7

Dataset from the Table 5.2 and 5.1 are saved in the csv file with semicolon as delimiter.

5.1. Fruits

First experiment is performed on the fruits dataset. File fruits3.csv contains fruits with two columns, namely vitamin C and A.

5.1.1. 2 features

The following command runs the algorithm:

```
python KMedoids.py -i fruits3.csv -c 2 --delimiter=";"
```

Several tests where performed which may be gathered in 3 different results presented on the Figure 5.1, 5.2, 5.3 together with clusters json.

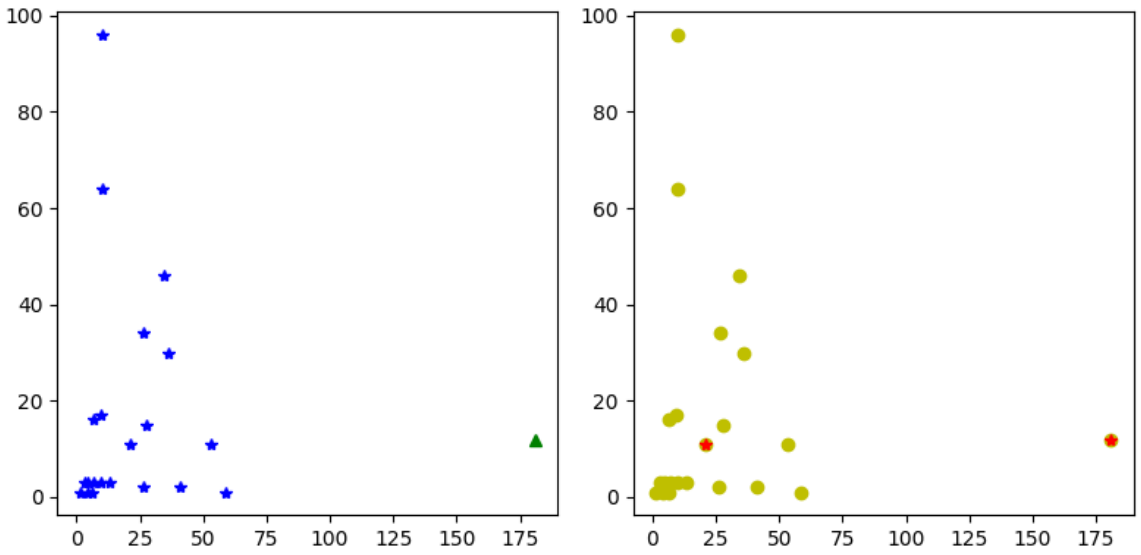

Figure 5.1. Fruits 3 result


```
{
    "Blackberry": [
        "Gooseberry",
        "Black lilac",
        "Blueberry",
        "Peach",
        "Sweet Cherry",
        "Pear",
        "Apple",
        "Blackberry",
        "Raspberry",
        "Apricot",
        "Red currant",
        "Plums",
        "Strawberry",
        "Grape",
        "Cherry",
        "Cranberry",
        "Oranges",
        "Grapefruit",
        "Tangerine",
        "Hazelnut",
        "Walnut"
    ],
    "Black currant": [
        "Black currant"
    ]
}
```


Table 5.2 contains vitamin and minerals of the fruits. The result obtained in the first experiment discovers one medoid black currant which has a lot of vitamin C, but little A

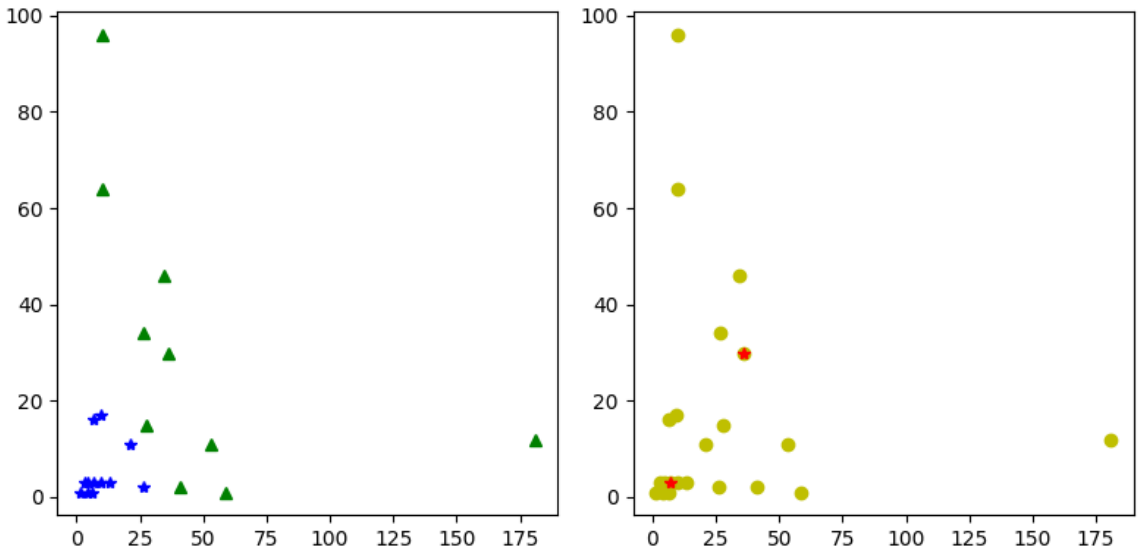

Figure 5.2. Fruits 3 result

```
{
    "Sweet Cherry": [
        "Blueberry",
        "Peach",
        "Sweet Cherry",
        "Pear",
        "Apple",
        "Blackberry",
        "Raspberry",
        "Plums",
        "Grape",
        "Cranberry",
        "Hazelnut",
        "Walnut"
    ],
    "Black lilac": [
        "Gooseberry",
        "Black lilac",
        "Apricot",
        "Black currant",
        "Red currant",
        "Strawberry",
        "Cherry",
        "Oranges",
        "Grapefruit",
        "Tangerine"
    ]
}
```

Next experiment divides the dataset into two clusters, first one grouped together close to the beginning of the coordinate system and further away. Two clusters reveals valuable fruits with a lot of vitamin C and/or A and second one poor of those vitamins.

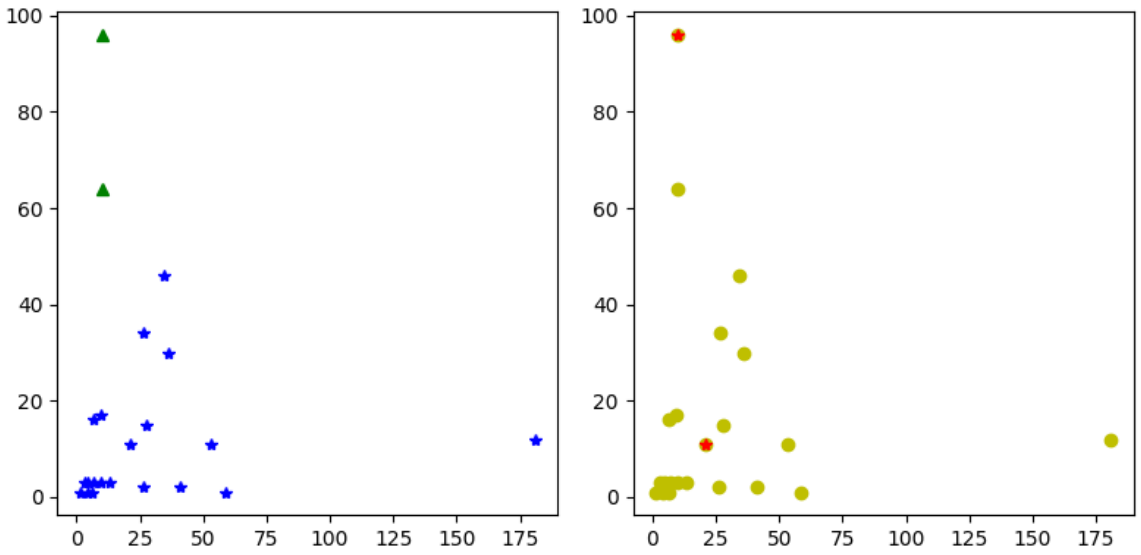

Figure 5.3. Fruits 3 result


```
{
    "Blackberry": [
        "Gooseberry",
        "Black lilac",
        "Blueberry",
        "Peach",
        "Sweet Cherry",
        "Pear",
        "Apple",
        "Blackberry",
        "Raspberry",
        "Black currant",
        "Red currant",
        "Plums",
        "Strawberry",
        "Grape",
        "Cranberry",
        "Oranges",
        "Grapefruit",
        "Tangerine",
        "Hazelnut",
        "Walnut"
    ],
    "Apricot": [
        "Apricot",
        "Cherry"
    ]
}
```


Last results shows items rich in vitamin A – Apricot and Cherry.

5.1.2. 3 features

Now vitamin K was added to the previous dataset. However visualisation affects only vitamin C and A, because the third one in 3 dimensions would be hardly readable. The following command runs the algorithm:

```
python KMedoids.py -i fruits2.csv -c 3 --delimiter=";"
```

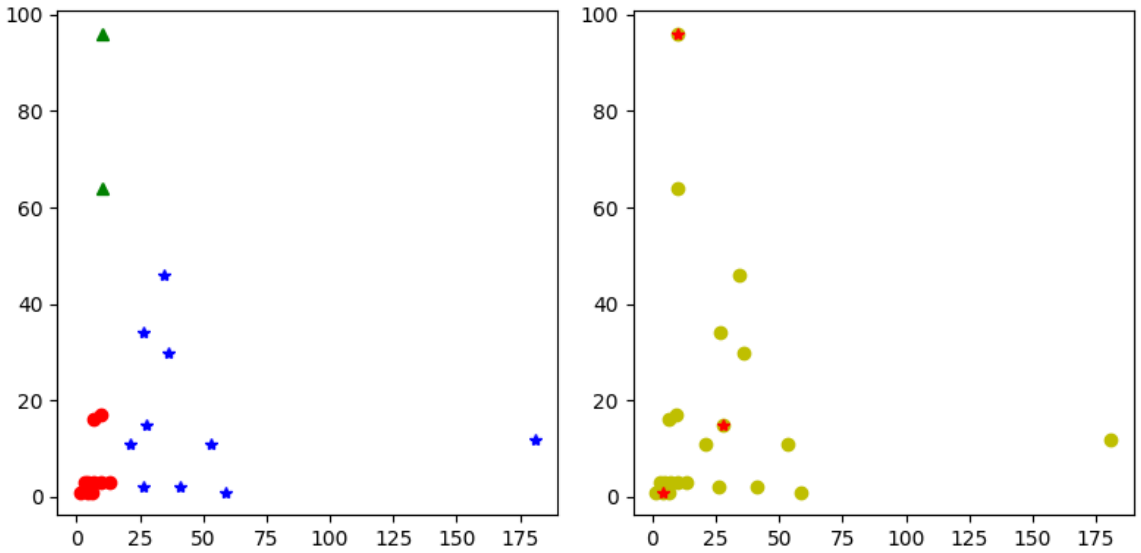

Figure 5.4. Fruits 2 result


```
{
    "Gooseberry": [
        "Gooseberry",
        "Black lilac",
        "Blackberry",
        "Raspberry",
        "Black currant",
        "Red currant",
        "Strawberry",
        "Oranges",
        "Grapefruit",
        "Tangerine"
    ],
    "Apricot": [
        "Apricot",
        "Cherry"
    ],
    "Pear": [
        "Blueberry",
        "Peach",
        "Sweet Cherry",
        "Pear",
        "Apple",
        "Plums",
        "Grape",
```

```
        "Cranberry",
        "Hazelnut",
        "Walnut"
    ]
}
```

Similarly to the previous result Apricot and Cherry were placed in one cluster. They are rich in vitamin A, the same amount of vitamin C – 10mg per 100g of flesh and contain vitamin K. On the other extreme pear represents fruits poor of vitamins.

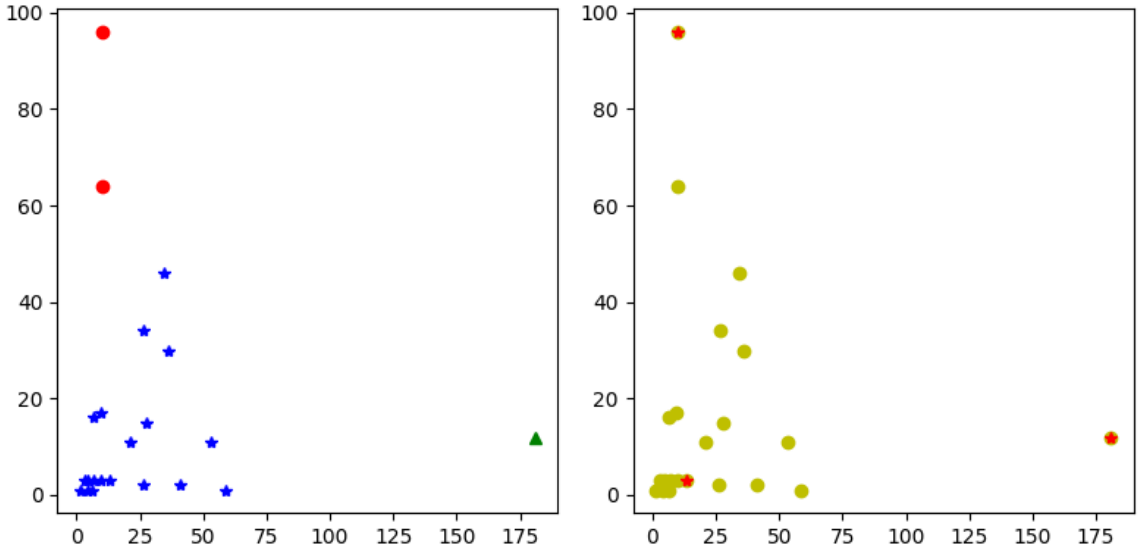

Figure 5.5. Fruits 2 result

```
{
    "Cranberry": [
        "Gooseberry",
        "Black lilac",
        "Blueberry",
        "Peach",
        "Sweet Cherry",
        "Pear",
        "Apple",
        "Blackberry",
        "Raspberry",
        "Red currant",
        "Plums",
        "Strawberry",
        "Grape",
        "Cranberry",
        "Oranges",
        "Grapefruit",
        "Tangerine",
        "Hazelnut",
        "Walnut"
    ],
```

```
    "Black currant": [
        "Black currant"
    ],
    "Apricot": [
        "Apricot",
        "Cherry"
    ]
}
```

Once again Apricot and Cherry are in the same cluster, black currant as an fruit with a lot of vitamin C some amount of A and no K in the cluster alone.

5.2. Cars

5.2.1. Nissans and BMWs 2 features

At the beginning Nissans and BMW were classifier into 2 clusters from the file cars1.csv. Only price and maximum speed were used.

```
python KMedoids.py -i cars1.csv -c 2 --delimiter=";"
```

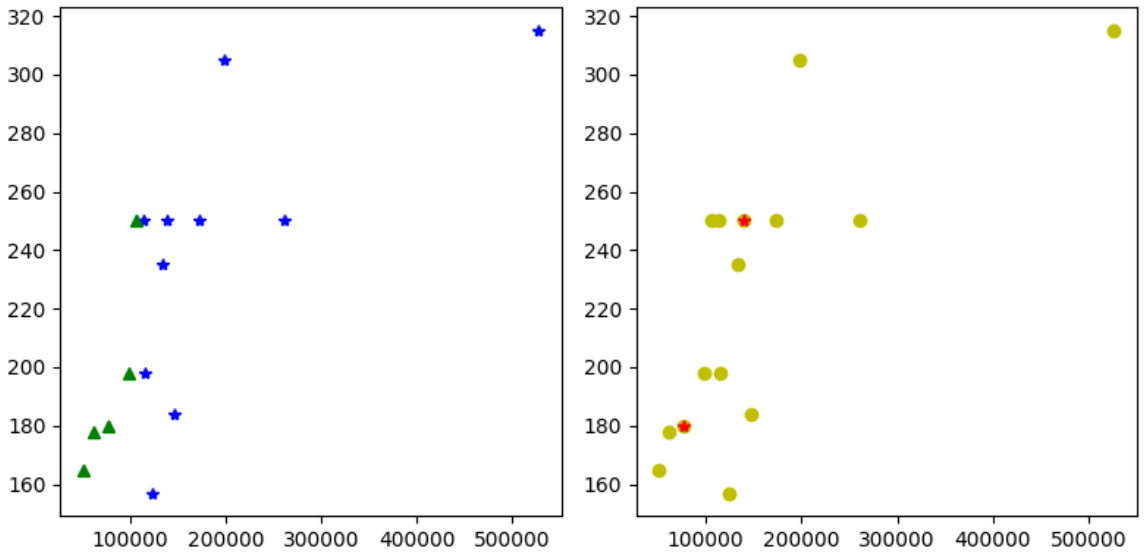

Figure 5.6. Cars 1 result

```
{
    "BMW X2": [
        "Nissan X-Trail",
        "Nissan Leaf",
        "Nissan Navara",
        "Nissan GT-R",
        "BMW Seria 2",
        "BMW X1",
        "BMW X2",
        "BMW Seria 4",
        "BMW Seria 5",
```

```
        "BMW Seria 6"
    ],
    "Nissan Juke": [
        "Nissan NV200",
        "Nissan Micra",
        "Nissan Juke",
        "Nissan Qashqai",
        "BMW Seria 1"
    ]
}
```

Classification may be interpreted as good and average cars. Good car is expensive and has decent maximum speed whereas bad cars are cheep and slow. Nissan Juke is an representative of average cars with trade off between speed and price. Surprisingly Nissan X-Trail was classified in as a good car instead of an average one.

5.2.2. Nissans and BMWs 3 features

Now third feature is added, which is acceleration.

```
python KMedoids.py -i cars6.csv -c 2 --delimiter=";"
```

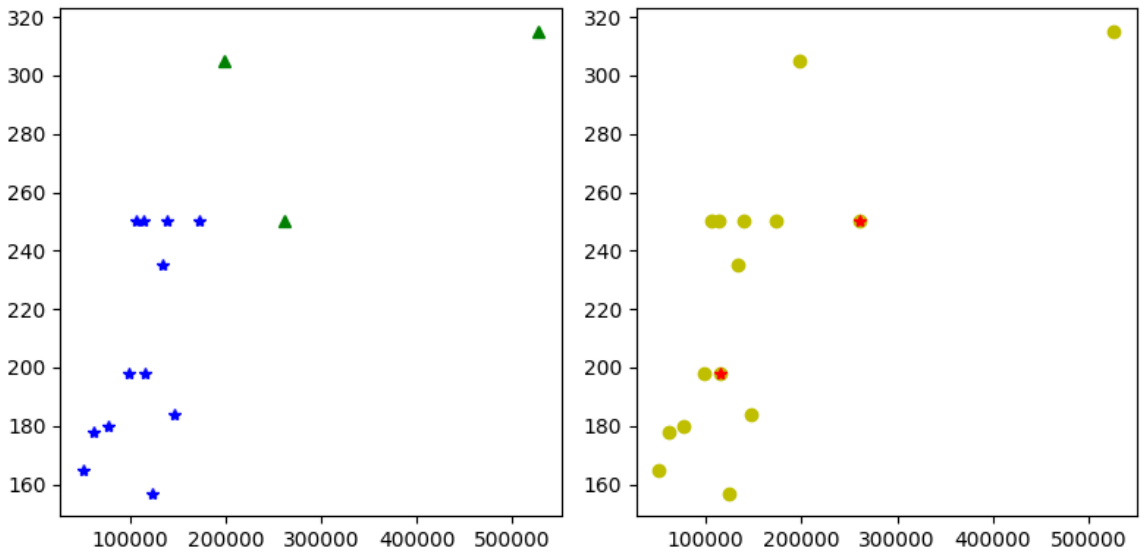

Figure 5.7. Cars 6 result

```
{
    "Nissan X-Trail": [
        "Nissan NV200",
        "Nissan Micra",
        "Nissan Juke",
        "Nissan Qashqai",
        "Nissan X-Trail",
        "Nissan Leaf",
        "Nissan Navara",
        "BMW Seria 1",
```

```
        "BMW Seria 2",
        "BMW X1",
        "BMW X2",
        "BMW Seria 4"
    ],
    "BMW Seria 6": [
        "Nissan GT-R",
        "BMW Seria 5",
        "BMW Seria 6"
    ]
}
```

Right now expensive and cars with great performance are represented by BMW Series 6, adding third feature improves the classification, because better cars are clearly in one cluster.

5.2.3. Nissans, BMWs, Dacias 3 features

In the last experiment all cars are classified with 3 features – maximal speed, price and acceleration.

```
python KMedoids.py -i cars3.csv -c 3 --delimiter=";"
```

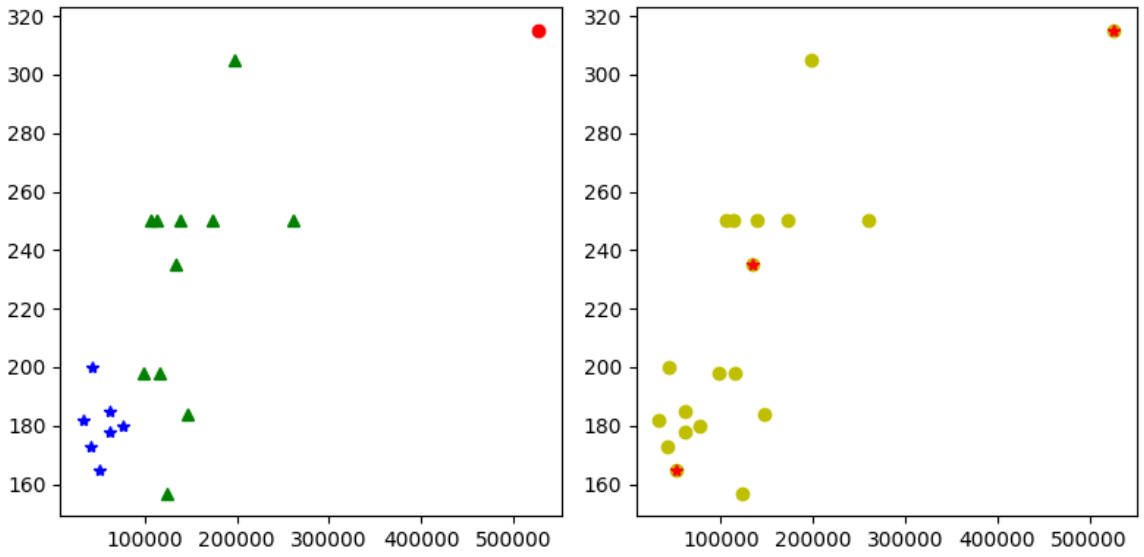

Figure 5.8. Cars 3 result

```
{
    "Nissan NV200": [
        "Nissan NV200",
        "Nissan Micra",
        "Nissan Juke",
        "Dacia Dokker",
        "Dacia Duster",
        "Dacia Lodgy",
        "Dacia Sandero"
    ],
```

```
    "BMW X1": [
        "Nissan Qashqai",
        "Nissan X-Trail",
        "Nissan Leaf",
        "Nissan Navara",
        "BMW Seria 1",
        "BMW Seria 2",
        "BMW X1",
        "BMW X2",
        "BMW Seria 4",
        "BMW Seria 5",
        "BMW Seria 6"
    ],
    "Nissan GT-R": [
        "Nissan GT-R"
    ]
}
```

PAM divides cars into 3 clusters – once again good interpretation is the average performance and price. Good, middle and bad cars are grouped separately very well. Nissan GT-R is clearly the best and the most expensive, all Dacias are in the same cluster which is logic because they are cheep and generally slow.

6. Conclusions

In this project PAM algorithm was implemented in Python as the command line script. Several datasets configurations were tested. At the beginning Fruits with vitamins and minerals juxtaposed with one another were classified. It revealed fruits rich in vitamin C or A. Consequently third vitamin was added, thereby fruits were divided into healthy with a lot of vitamins and poor once. From the point of view of amount of vitamin C, K and A it is better to eat apricot or cherry than pear.

Finally cars were classified into 2 and 3 clusters. Dataset comes from legitimate resource, so PAM algorithm may help to choose the car with a trade off between speed and price. Slow cars are generally cheep – all Dacias were placed in the one cluster. Nissan GTR is fast but extremelly expensive what was confirmed by the algorithm.

Data classification into clusters with a help of PAM algorithm is a useful data mining algorithm, which reveals information not visible at the first glance.

Bibliography

[1] Wikipedia, The Free Encyclopedia. K-medoids. https://en.wikipedia.org/wiki/K-medoids. [Accessed 10.01.2021].

List of Figures

List of Tables