- Author: Juan Antonio
- Last Update: 2021-02-07

0.- A brief introduction to Clustering Algorithms

Clustering is a type of *Unsupervised Learning Algorithm*, that means that **there are no variable to predict or forecast attached to the data**. There is no ouput, just attributes that describes the data. The goal is to group similar instances together into clusters.

Although clustering doesn't predict or forecast anything, it is a great tool for data analysis, customer segmentation, recommender systems, search engines, image segmentation, semi-supervised learning, dimensionality reduction, and more.

Clustering is used in a wide variety of applications, including:

- For customer segmentation: you can cluster your customers based on their purchases, their activity on your website, and so on. For example, this can be useful in *recommender systems*.
- For data analysis: when analyzing a new dataset, it is often useful to first discover clusters of similar instances, as it is often easier to analyze clusters separately.
- As a dimensionality reduction technique: once a dataset has been clustered, it is usually possible to measure each instance's affinity with each cluster.
- For anomaly detection (also called outlier detection): any instance that has a low affinity to all the clusters is likely to be an anomaly.
- For semi-supervised learning: if you only have a few labels, you could perform clustering and propagate the labels to all the instances in the same cluster.
- For search engines: for example, some search engines let you search for images that are similar to a reference image.
- To segment an image: by clustering pixels according to their color, then replacing each pixel's color with the mean color of its cluster, it is possible to reduce the number of different colors in the image considerably.

Within clustering algorithms, there're several models that might be applied:

Method name	Parameters	Scalability	Usecase	Geometry (metric used)
K-Means	number of clusters	Very large `n samples`, medium `n_clusters` with MiniBatch code	General-purpose, even cluster size, flat geometry, not too many clusters	Distances between points
Affinity propagation	damping, sample preference	Not scalable with n_samples	Many clusters, uneven cluster size, non-flat geometry	Graph distance (e.g. nearest-neighbor graph)
Mean-shift	bandwidth	Not scalable with `n_samples`	Many clusters, uneven cluster size, non-flat geometry	Distances between points
Spectral clustering	number of clusters	Medium `n samples`, small `n_clusters`	Few clusters, even cluster size, non-flat geometry	Graph distance (e.g. nearest-neighbor graph)
Ward hierarchical clustering	number of clusters or distance threshold	Large `n samples` and `n_clusters`	Many clusters, possibly connectivity constraints	Distances between points
Agglomerative clustering	number of clusters or distance threshold, linkage type, distance	Large `n samples` and `n_clusters`	Many clusters, possibly connectivity constraints, non Euclidean distances	Any pairwise distance
DBSCAN	neighborhood size	Very large `n samples`, medium `n_clusters`	Non-flat geometry, uneven cluster sizes	Distances between nearest points
OPTICS	minimum cluster membership	Very large `n samples`, large `n_clusters`	Non-flat geometry, uneven cluster sizes, variable cluster density	Distances between points
Gaussian mixtures	many	Not scalable	Flat geometry, good for density estimation	Mahalanobis distances to centers
Birch	branching factor, threshold, optional global clusterer.	Large `n clusters` and `n_samples`	Large dataset, outlier removal, data reduction.	Euclidean distance between points

For the next problem, I will study the following ones:

- **K-means**: General-purpose, even cluster size, flat geometry, not too many clusters.
- **Hierarchical Clustering**: Very large n_samples, medium n_clusters.
- **Density Based Clustering (DBSCAN)**:Very large n_samples, medium n_clusters.
- **Mean shift**: Very large n_samples, medium n_clusters.

In [1]:
```
import numpy as np
import pandas as pd
import warnings
warnings.filterwarnings('ignore')

# Visualization
import plotly.express as px
import plotly.figure_factory as ff
from plotly.subplots import make_subplots
import plotly.graph_objects as go
%matplotlib inline
from matplotlib import pyplot as plt
import seaborn as sns


# Tools for predictive data analysis
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
from sklearn.cluster import AgglomerativeClustering
from scipy.cluster import hierarchy
from scipy.spatial import distance_matrix
from sklearn.cluster import DBSCAN
from sklearn.cluster import MeanShift, estimate_bandwidth
```

1.- The Mall Customer Segmentation problem

The `Mall Customer Segmentation Data` dataset is created only for the learning purpose of the customer segmentation concepts, also known as market basket analysis.

Content

You are owing a supermarket mall and through membership cards, you have some basic data about your customers like Customer ID, age, gender, annual income and spending score (spending Score is something you assign to the customer based on your defined parameters like customer behavior and purchasing data).

In [2]:
```
df = pd.read_csv('../input/customer-segmentation-tutorial-in-python/Mall_Customers.csv')
df.columns = ['customer_id','gender','age', 'income','score']
display(df.head())
df.describe()
```

	customer_id	gender	age	income	score
0	1	Male	19	15	39
1	2	Male	21	15	81
2	3	Female	20	16	6
3	4	Female	23	16	77
4	5	Female	31	17	40

Out[2]:

	customer_id	age	income	score
count	200.000000	200.000000	200.000000	200.000000
mean	100.500000	38.850000	60.560000	50.200000
std	57.879185	13.969007	26.264721	25.823522
min	1.000000	18.000000	15.000000	1.000000
25%	50.750000	28.750000	41.500000	34.750000
50%	100.500000	36.000000	61.500000	50.000000
75%	150.250000	49.000000	78.000000	73.000000
max	200.000000	70.000000	137.000000	99.000000

1.1.- KNN-Means

K-means is often referred to as Lloyd's algorithm (*Stuart Lloyd at the Bell Labs, 1957* (https://en.wikipedia.org/wiki/Lloyd%27s_algorithm))

- Most popular clustering model. Simple and fast.
- KNN scales well to large number of samples and has been used across a large range of application areas in many different fields.

The **KMeans** algorithm clusters data by trying to separate samples in **n** groups of equal variance, minimizing a criterion known as the inertia or within-cluster sum-of-squares (see below). This algorithm requires the number of clusters to be specified.

- The k-means algorithm divides a set of N samples X into K disjoint clusters C, each described by the mean μ_j of the samples in the cluster.
- The means are commonly called the cluster "centroids"; note that they are not, in general, points from X, although they live in the same space.

The K-means algorithm aims to choose centroids that minimise the inertia, or within-cluster sum-of-squares criterion.

Method name	Parameters	Scalability	Usecase	Geometry (metric used)
K-Means	number of clusters	Very large `n samples`, medium `n_clusters` with MiniBatch code	General-purpose, even cluster size, flat geometry, not too many clusters	Distances between points
Affinity propagation	damping, sample preference	Not scalable with n_samples	Many clusters, uneven cluster size, non-flat geometry	Graph distance (e.g. nearest-neighbor graph)
Mean-shift	bandwidth	Not scalable with `n_samples`	Many clusters, uneven cluster size, non-flat geometry	Distances between points
Spectral clustering	number of clusters	Medium `n samples`, small `n_clusters`	Few clusters, even cluster size, non-flat geometry	Graph distance (e.g. nearest-neighbor graph)
Ward hierarchical clustering	number of clusters or distance threshold	Large `n samples` and `n_clusters`	Many clusters, possibly connectivity constraints	Distances between points
Agglomerative clustering	number of clusters or distance threshold, linkage type, distance	Large `n samples` and `n_clusters`	Many clusters, possibly connectivity constraints, non Euclidean distances	Any pairwise distance
DBSCAN	neighborhood size	Very large `n samples`, medium `n_clusters`	Non-flat geometry, uneven cluster sizes	Distances between nearest points
OPTICS	minimum cluster membership	Very large `n samples`, large `n_clusters`	Non-flat geometry, uneven cluster sizes, variable cluster density	Distances between points
Gaussian mixtures	many	Not scalable	Flat geometry, good for density estimation	Mahalanobis distances to centers
Birch	branching factor, threshold, optional global clusterer.	Large `n clusters` and `n_samples`	Large dataset, outlier removal, data reduction.	Euclidean distance between points

Note: Inertia can be recognized as a measure of how internally coherent clusters are.

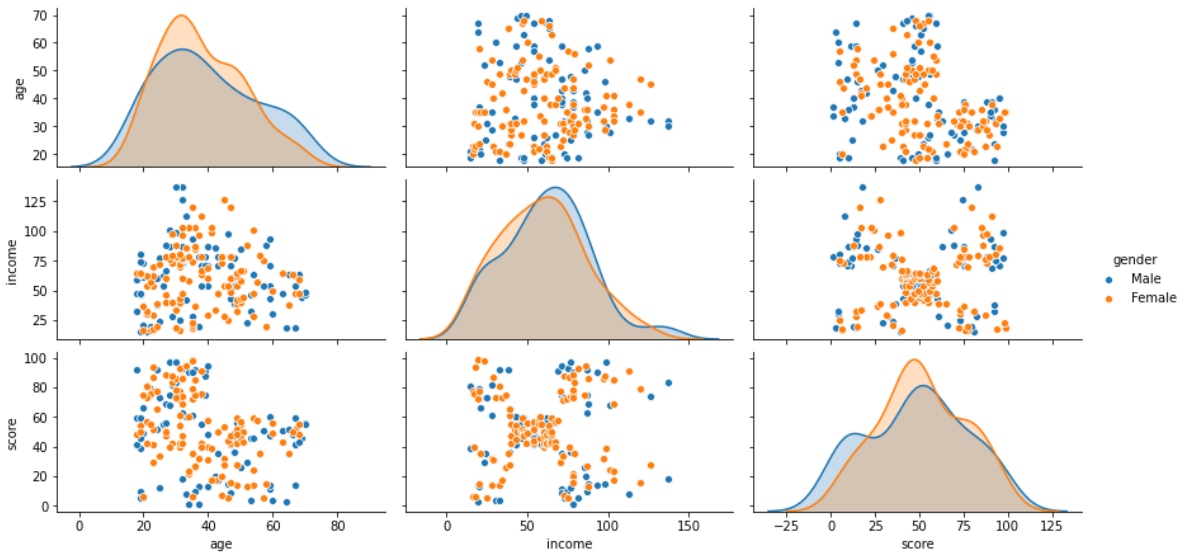
In summary, this algorithm has three steps:

1. Chooses the initial centroids, with the most basic method being to choose k samples from the dataset X.
2. Looping between the following steps:
    - Assigns each sample to its nearest centroid.
    - Creates new centroids by taking the mean value of all of the samples assigned to each previous centroid.

The difference between the old and the new centroids are computed and the algorithm repeats these last two steps until this value is less than a threshold. In other words, it repeats until the centroids do not move significantly.

Let's see if gender is a good feature for segmenting the users in the previous dataset. As we can see in the next chart, there isn't a direct relation in order to segment the customers using this feature. However is possible to see some interesting patterns like the relation between `Income` and `Score` or `Age` and `Score`

In [3]:
```
df_plot = df.drop(['customer_id'], axis=1)
# px.scatter_matrix(df_plot, dimensions=['age', 'income','score'], color='gender')
sns.pairplot(df_plot, hue='gender', height=2, aspect=2);
```



We don't need the `gender` attribute neither `customer_id` for segmenting customers so let's drop it and study the other features.

In [4]:
```
df2 = df.drop(['customer_id','gender'], axis=1)
```

In [5]:
```
n_clusters = []
res_clusters = []

for i in range (1, 11):
    knn_model = KMeans(n_clusters=i).fit(df2)
    n_clusters.append(i)
    res_clusters.append(knn_model.inertia_) # inertia_: Sum of squared distances of samp
les to their closest cluster center.

df_knn = pd.DataFrame(data= {'n_clusters': n_clusters, 'res_clusters':res_clusters})
```

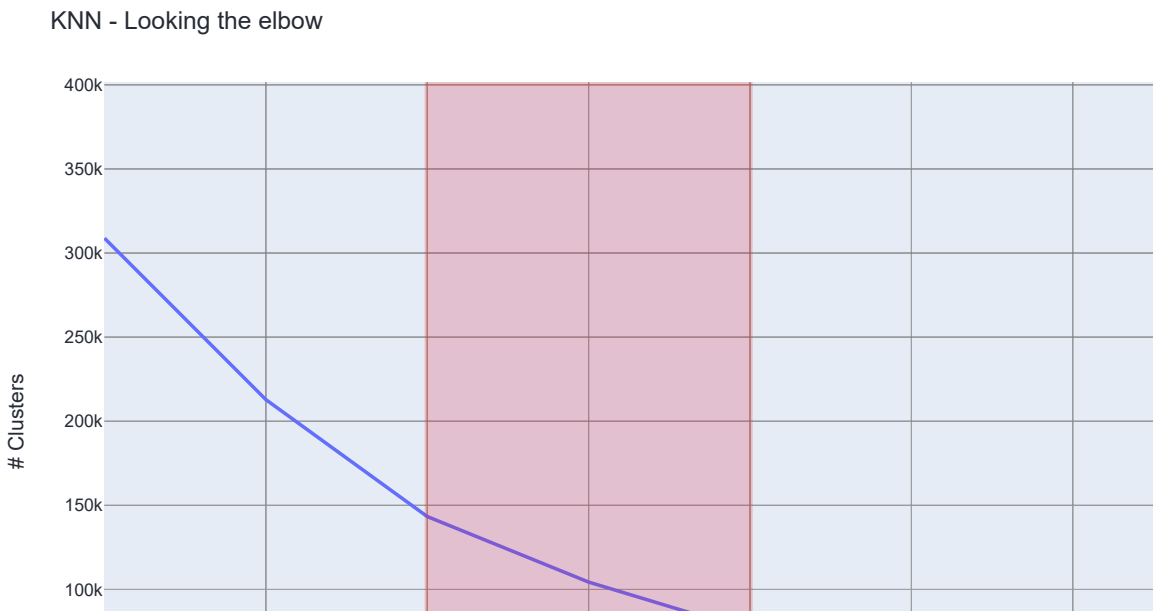
We have created several KNN models with different number of clusters (from 1 to 10). We have to choose a number cluster in a way that:

- The sum of squared distances of samples to their closest cluster center (`inertia_`) is minimum as possible.
- The number of clusters is maximum of possible (we want to group the instances in clusters with the biggest size as possible). This is known as *searching the elbow*.

Looking the next chart, we are looking a number **between 3 and 5 clusters**.

In [6]:
```
fig = px.line(df_knn, x="n_clusters", y="res_clusters")
fig.add_shape(type="rect", xref="x", yref="y", x0=3, y0=400*1000, x1=5, y1=1,line=dict(c
olor="red",width=3,), fillcolor="crimson", opacity=0.2)
fig.update_layout(
    title="KNN - Looking the elbow",
    xaxis_title="intertia", yaxis_title="# Clusters",
    font=dict(family="Arial", size=10, color="#262b35")
)

fig.show()
```



Looking the next plots, seems that the best choice is using **5 clusters** (we can't know for sure if this is the best way of labeling the data due to this is a *unsupervised problem* though). So with this grouping we can say that:

- Label 0 is low income and low spending
- Label 1 is high income and high spending
- Label 2 is mid income and mid spending
- Label 3 is high income and low spending
- Label 4 is low income and high spending

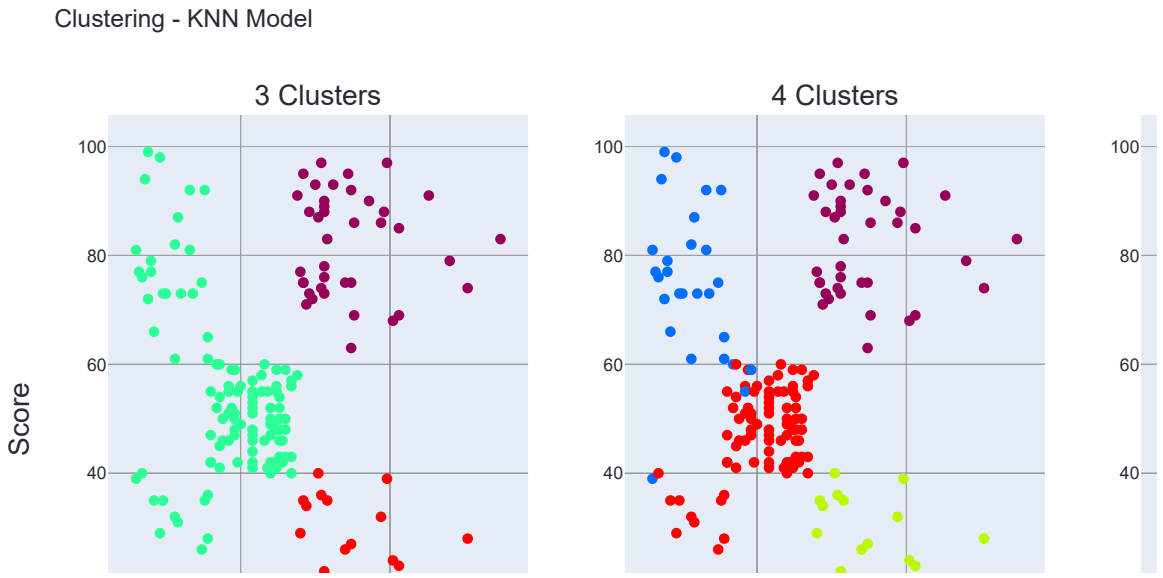
In [7]:
```
def knn_visuals(dataset, nc):
    model = KMeans(n_clusters=nc).fit(dataset)
    dataset['labels'] = model.labels_
    return (dataset)

df_knn3,df_knn4,df_knn5 = df2.copy(),df2.copy(),df2.copy()

knn3 = knn_visuals(df_knn3, 3)
knn4 = knn_visuals(df_knn4, 4)
knn5 = knn_visuals(df_knn5, 5)
```

In [8]:
```
fig = make_subplots(rows=1, cols=3 ,x_title='Income',y_title='Score', subplot_titles=("3 Clusters", "4 Clusters", "5 Clusters"))
fig.add_trace(go.Scatter(x=knn3['income'], y=knn3['score'], mode='markers', marker=dict(colorscale='rainbow'), marker_color=knn3['labels'], text=knn3['labels']), row=1, col=1)
fig.add_trace(go.Scatter(x=knn4['income'], y=knn4['score'], mode='markers', marker=dict(colorscale='rainbow'), marker_color=knn4['labels'], text=knn4['labels']), row=1, col=2)
fig.add_trace(go.Scatter(x=knn5['income'], y=knn5['score'], mode='markers', marker=dict(colorscale='rainbow'), marker_color=knn5['labels'], text=knn5['labels']), row=1, col=3)
fig.update_layout(
    title="Clustering - KNN Model",
    font=dict(family="Arial", size=10, color="#262b35"),
    showlegend=False
)

fig.show()
```



In [9]:
```
fig = make_subplots(rows=1, cols=2 ,x_title='Label', subplot_titles=("Clustering for Income", "Clustering for Score"))

labels = sorted(knn5.labels.unique())

for l in labels:
    fig.add_trace(go.Violin(x=knn5['labels'][knn5['labels'] == l], y=knn5['income'][knn5['labels'] == l], name=str(l), box_visible=True, meanline_visible=True),row=1, col=1)
    fig.add_trace(go.Violin(x=knn5['labels'][knn5['labels'] == l], y=knn5['score'][knn5['labels'] == l], name=str(l), box_visible=True, meanline_visible=True),row=1, col=2)

fig.update_layout(
    title="Clustering - Attributes per Cluster",
    font=dict(family="Arial", size=10, color="#262b35"),
    showlegend=False
)

fig.show()
```

Hierarchical Clustering

Agglomerative Hierarchical Clustering

Hierarchical clustering is a general family of clustering algorithms that build nested clusters by merging or splitting them successively. This hierarchy of clusters is represented as a tree (or dendrogram). The root of the tree is the unique cluster that gathers all the samples, the leaves being the clusters with only one sample.

AgglomerativeClustering can also scale to large number of samples when it is used jointly with a connectivity matrix, but is computationally expensive when no connectivity constraints are added between samples: it considers at each step all the possible merges.

This is a type of clustering that requires two types of inputs:

- `n_clusters` : Number of clusters or centroids to generate.
- `linkage` : Which linkage criterion to use. The linkage criterion determines which distance to use between sets of observation. The algorithm will merge the pairs of cluster that minimize this criterion.:
  - ‘ward’ minimizes the variance of the clusters being merged.
  - ‘average’ uses the average of the distances of each observation of the two sets.
  - ‘complete’ or ‘maximum’ linkage uses the maximum distances between all observations of the two sets.
  - ‘single’ uses the minimum of the distances between all observations of the two sets.

Note: distance matrix contains the distance from each point to every other point of a dataset.

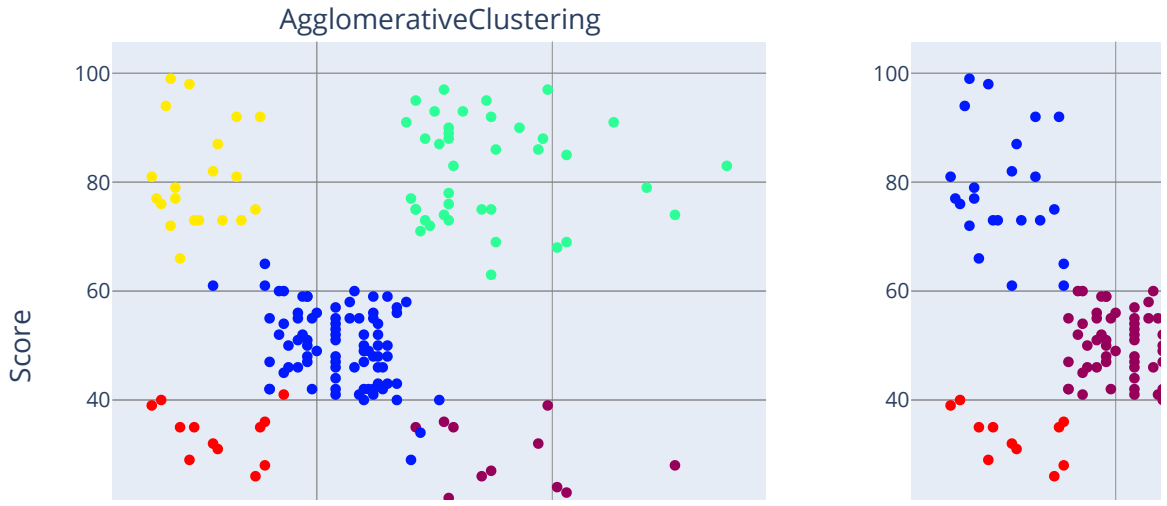
In [10]:
```
linkages = ['ward','average','complete','single']

def apply_AgglomerativeClustering(df, n, l):
    model = AgglomerativeClustering(n_clusters=5, linkage='average').fit(df)
    df['labels'] = model.labels_
    return (df)


df_ward_plot = apply_AgglomerativeClustering(df2.copy(), 5, linkages[0])
df_avg_plot = apply_AgglomerativeClustering(df2.copy(), 5, linkages[1])
df_com_plot = apply_AgglomerativeClustering(df2.copy(), 5, linkages[2])
df_sin_plot = apply_AgglomerativeClustering(df2.copy(), 5, linkages[3])



fig = make_subplots(rows=1, cols=2 ,x_title='Income',y_title='Score', subplot_titles=("A
gglomerativeClustering", "KNN"))
fig.add_trace(go.Scatter(x=df_ward_plot['income'], y=df_ward_plot['score'], mode='marker
s', marker=dict(colorscale='rainbow'), marker_color=df_ward_plot['labels'], text=df_ward
_plot['labels']), row=1, col=1)
fig.add_trace(go.Scatter(x=knn5['income'], y=knn5['score'], mode='markers', marker=dict(
colorscale='rainbow'), marker_color=knn5['labels'], text=knn5['labels']), row=1, col=2)
fig.update_layout(showlegend=False)
```

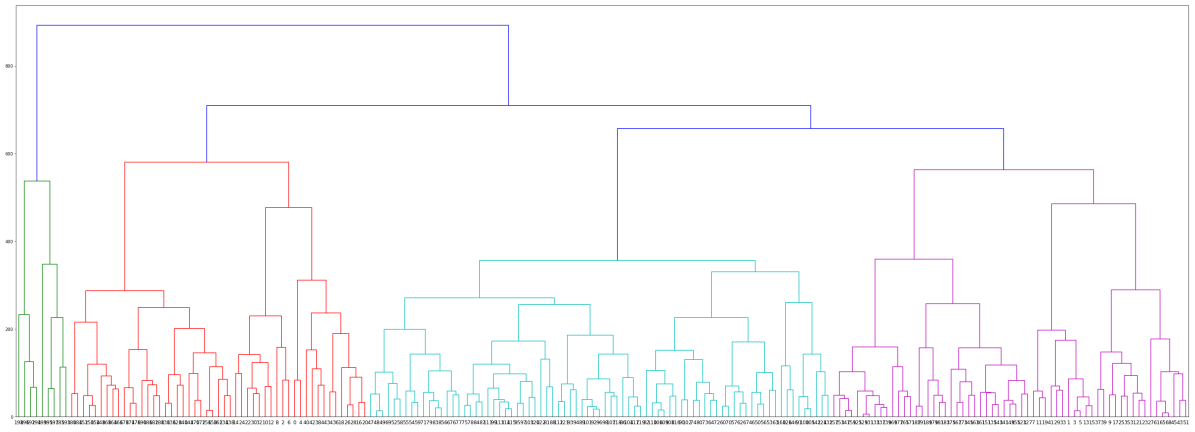


Agglomerative Hierarchical Clustering - Dendogram

It’s possible to visualize the tree representing the hierarchical merging of clusters as a dendrogram. Visual inspection can often be useful for understanding the structure of the data, though more so in the case of small sample sizes.

- It’s possible to get the **distance matrix** that contains the distance from each point to every other point of a dataset. Will be useful for using it with the **linkage** class.
- Using the [hierarchy.linkage (https://docs.scipy.org/doc/scipy/reference/cluster.hierarchy.html)](https://docs.scipy.org/doc/scipy/reference/cluster.hierarchy.html) function, we'll be able to perform hierarchical/agglomerative clustering. This function cut hierarchical clusterings into flat clusterings or find the roots of the forest formed by a cut by providing the flat cluster ids of each observation.

In [11]:
```
# Get distance matrix
distance = distance_matrix(df2, df2)
# Get hierarchy
Z = hierarchy.linkage(distance, 'complete');
plt.figure(figsize=(50, 18))
hierarchy.dendrogram(Z, leaf_rotation=0, leaf_font_size=12, orientation='top');
```



Density Based Clustering (DBSCAN)

K-means, hierarchical and fuzzy clustering perform really well in non-supervised data, however, for tasks with arbitrary shape clusters or clusters within clusters, those techniques might perform poorly (that's because elements in the same cluster might not share similarities).

The DBSCAN algorithm views clusters as areas of high density separated by areas of low density. Due to this rather generic view, **clusters found by DBSCAN can be any shape**, as opposed to k-means which assumes that clusters are convex shaped.

The central component to the DBSCAN is the concept of core samples, which are samples that are in areas of high density. A cluster is therefore a set of core samples, each close to each other (measured by some distance measure) and a set of non-core samples that are close to a core sample (but are not themselves core samples).

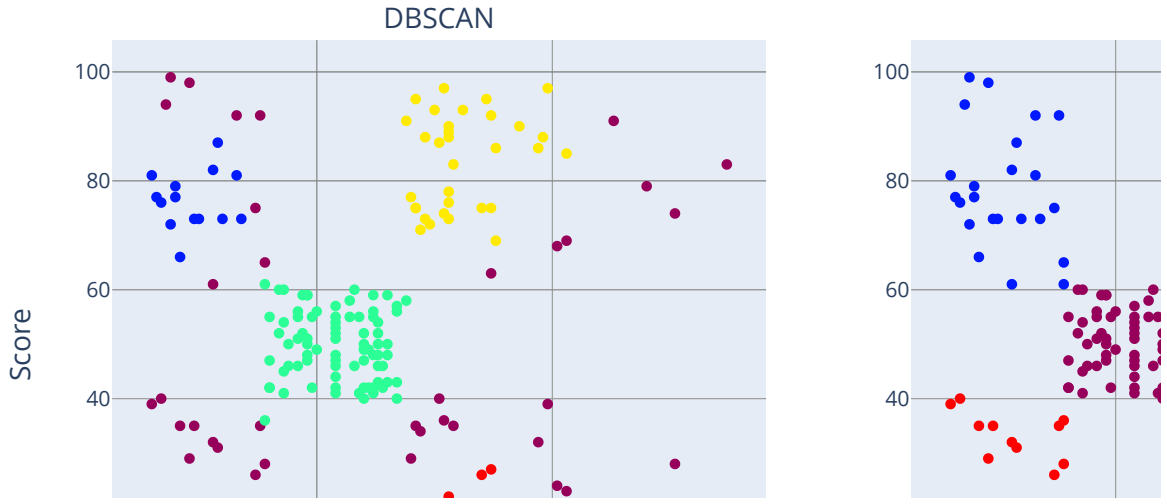
The whole idea is that if a particular point belongs to a cluster, it should be near to lots of other points in that cluster.Note: **Density** = Number of points within a specified radius.

DBSCAN uses two different parameters:

- **Epsilon**: Determines a specified radius that if includes enough number of points within, we call it dense area.
- **minimumSamples**: Determines the minimum number of data points we want in a neighborhood to define a cluster.

In [12]:
```
df_dbscan = df2.copy()
model = DBSCAN(eps=11, min_samples=6).fit(df_dbscan)
df_dbscan['labels'] = model.labels_

fig = make_subplots(rows=1, cols=2 ,x_title='Income',y_title='Score', subplot_titles=("D
BSCAN", "KNN"))
fig.add_trace(go.Scatter(x=df_dbscan['income'], y=df_dbscan['score'], mode='markers', ma
rker=dict(colorscale='rainbow'), marker_color=df_dbscan['labels'], text=df_dbscan['label
s']), row=1, col=1)
fig.add_trace(go.Scatter(x=knn5['income'], y=knn5['score'], mode='markers', marker=dict(
colorscale='rainbow'), marker_color=knn5['labels'], text=knn5['labels']), row=1, col=2)
fig.update_layout(showlegend=False)
```



Is possible to see that DBSCAN doesn't perform well. **The density in the data is not strong enough** (we can see that we have labels -1).

Mean Shift Algorithm

Centroid based algorithm which goal is finding blobs in a smooth density of samples. It works as next:

1. Updating candidates for centroids to be the mean of the points withing a given region.
2. Those candidates are filtered in a post-processing stage to eliminate near-duplicates to form the final set of centroids.
3. The algorithm sets the number of clusters (instead of relying on a paremeter bandwith that dictates the size of the region for searching).

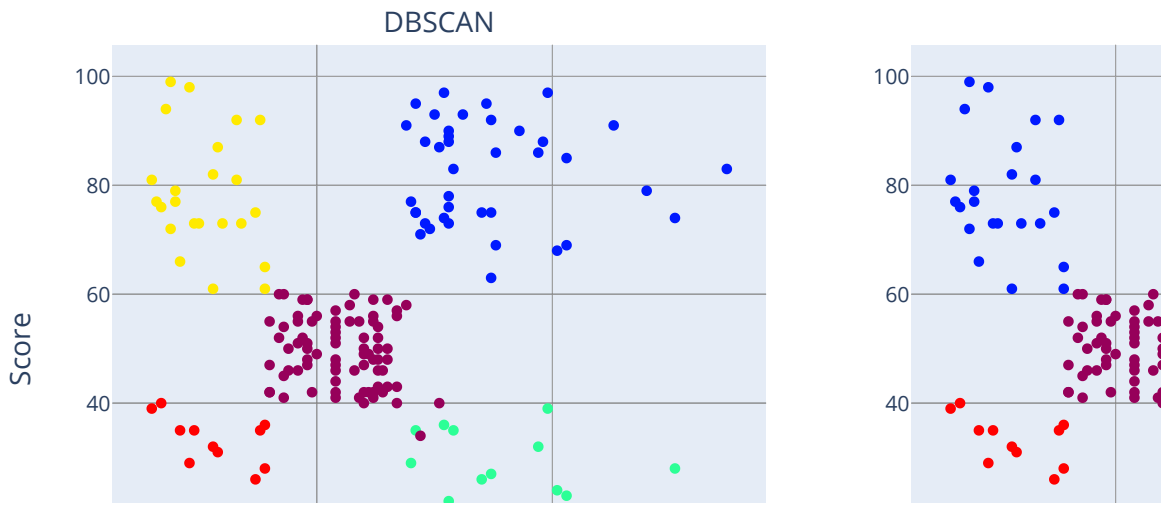
The central component to the DBSCAN is the concept of core samples, which are samples that are in areas of high density. A cluster is therefore a set of core samples, each close to each other (measured by some distance measure) and a set of non-core samples that are close to a core sample (but are not themselves core samples).

The algorithm is not highly scalable, as it requires multiple nearest neighbor searches during the execution of the algorithm. The algorithm is guaranteed to converge, however the algorithm will stop iterating when the change in centroids is small.

In [13]:
```
df_mean = df2.copy()


bandwidth = estimate_bandwidth(df_mean, quantile=0.1)
model = MeanShift(bandwidth).fit(df_mean)
df_mean['labels'] = model.labels_

fig = make_subplots(rows=1, cols=2 ,x_title='Income',y_title='Score', subplot_titles=("D
BSCAN", "KNN"))
fig.add_trace(go.Scatter(x=df_mean['income'], y=df_mean['score'], mode='markers', marker
=dict(colorscale='rainbow'), marker_color=df_mean['labels'], text=df_mean['labels']), ro
w=1, col=1)
fig.add_trace(go.Scatter(x=knn5['income'], y=knn5['score'], mode='markers', marker=dict(
colorscale='rainbow'), marker_color=knn5['labels'], text=knn5['labels']), row=1, col=2)
fig.update_layout(showlegend=False)
```

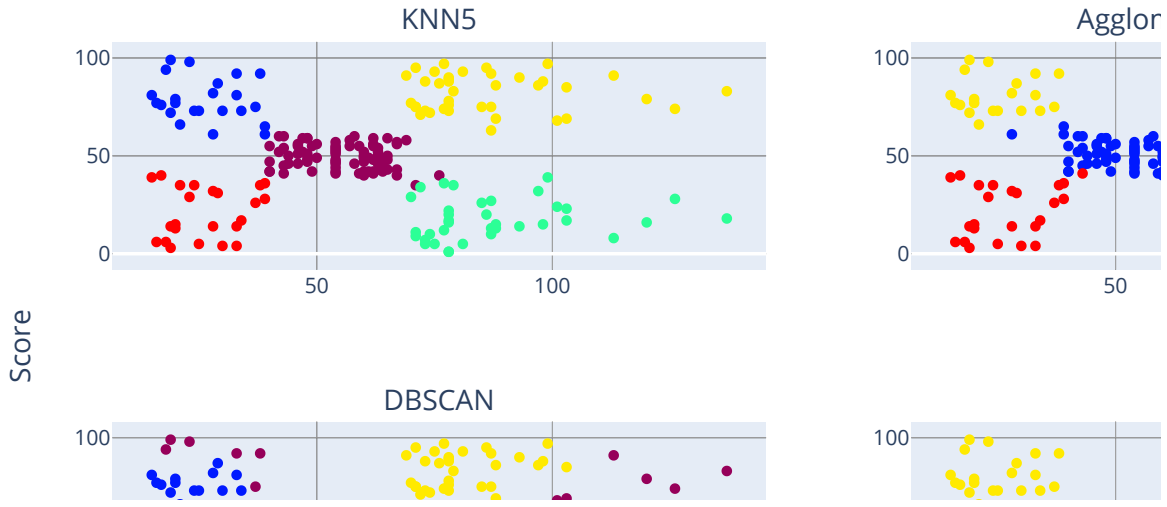


Summary

- Most of the algorithms use 5 clusters (except DBSCAN).
- KNN5, AgglomerativeClustering and Mean Shift perform really good for clustering the instances while DBSCAN not.
- DBSCAN is not performing well because the density is not enough for feeding this algorithm.

In [14]:
```
fig = make_subplots(rows=2, cols=2 ,x_title='Income',y_title='Score', subplot_titles=("K
NN5", "AgglomerativeClustering", "DBSCAN", "Mean Shift"))


fig.add_trace(go.Scatter(x=knn5['income'], y=knn5['score'], mode='markers', marker=dict(
colorscale='rainbow'), marker_color=knn5['labels'], text=knn5['labels']), row=1, col=1)
fig.add_trace(go.Scatter(x=df_ward_plot['income'], y=df_ward_plot['score'], mode='marker
s', marker=dict(colorscale='rainbow'), marker_color=df_ward_plot['labels'], text=df_ward
_plot['labels']), row=1, col=2)
fig.add_trace(go.Scatter(x=df_dbscan['income'], y=df_dbscan['score'], mode='markers', ma
rker=dict(colorscale='rainbow'), marker_color=df_dbscan['labels'], text=df_dbscan['label
s']), row=2, col=1)
fig.add_trace(go.Scatter(x=df_mean['income'], y=df_mean['score'], mode='markers', marker
=dict(colorscale='rainbow'), marker_color=df_mean['labels'], text=df_mean['labels']), ro
w=2, col=2)
fig.update_layout(showlegend=False)
```



Bibliography

- Scikit-learn (clustering) (https://scikit-learn.org/stable/modules/clustering.html)
- Popular Unsupervised Clustering Algorithms (https://www.kaggle.com/fazilbtopal/popular-unsupervised-clustering-algorithms)
- Hands on Machine Learning (Unsupervised Learning Techniques) (https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&cad=rja&uact=8&ved=2ahUKEwiA28ezztfuAhVLUxUIHbK9B3IQFjAAegQIAhAC
Machine-Learning-Scikit-Learn-TensorFlow%2Fdp%2F1491962291&usg=AOvVaw1q1iijqnY15HH6sH-kvWun)
- Mall Customer Segmentation Data (https://www.kaggle.com/vjchoudhary7/customer-segmentation-tutorial-in-python)