Importing neccessary libraries

In [1]:
```
import numpy as np
import math
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from sklearn.cluster import KMeans
```

Pre-processing the data

The dataset is taken from banknote-authentication dataset provided by Volker Lohweg (University of Applied Sciences, Ostwestfalen-Lippe)

In the dataset we only take 2 features: V1 (Variance of Wavelet Transformed image), V2 (Skewness of Wavelet Transformed image)

In [75]:
```
dt=pd.read_csv('BankNotes.csv')
dt.head()
```

Out[75]:

	V1	V2
0	3.62160	8.6661
1	4.54590	8.1674
2	3.86600	-2.6383
3	3.45660	9.5228
4	0.32924	-4.4552

In [76]:
```
rows, cols=dt.shape[0], dt.shape[1]
means=[]
stds=[]
for i in range(cols):
    vCol=dt[f'V{i+1}']
    print(f'V{i+1} \n Range: {vCol.max()-vCol.min()}, Mean: {vCol.mean()}, Std: {vC
    means.append(vCol.mean())
    stds.append(vCol.std())
```

```
V1
 Range: 13.8669, Mean: 0.43373525728862977, Std: 2.8427625862451658
V2
 Range: 26.7247, Mean: 1.9223531209912539, Std: 5.869046743580378
```

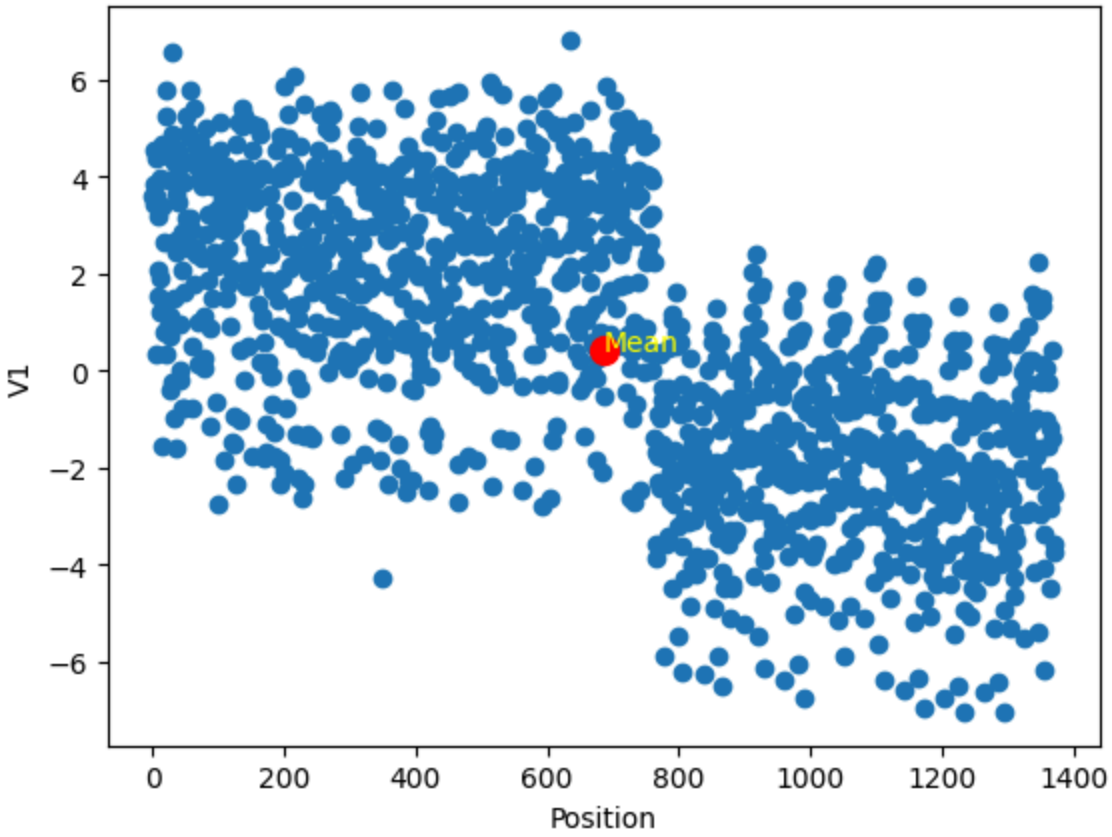
We saw that the ***range*** of the values in two columns is *significantly different*, resulting in *variations* in the ***mean*** of those columns. Additionally, the ***standard deviations*** in these two columns are *large*. Given this data, applying **K-means clustering** without normalization

could yield misleading results due to the disparity in the two columns. Therefore, it's essential to ***normalize*** the data before applying K-means. Before to normalization, we aim to visualize the data to gain a better understanding of its distribution.

- Visually V1

In [77]:
```
plt.xlabel("Position")
plt.ylabel("V1")
plt.scatter([i for i in range(rows)], dt['V1'])
plt.scatter(rows/2, means[0], color='red', s=100)
plt.text(rows/2, means[0], 'Mean', color='yellow')
```

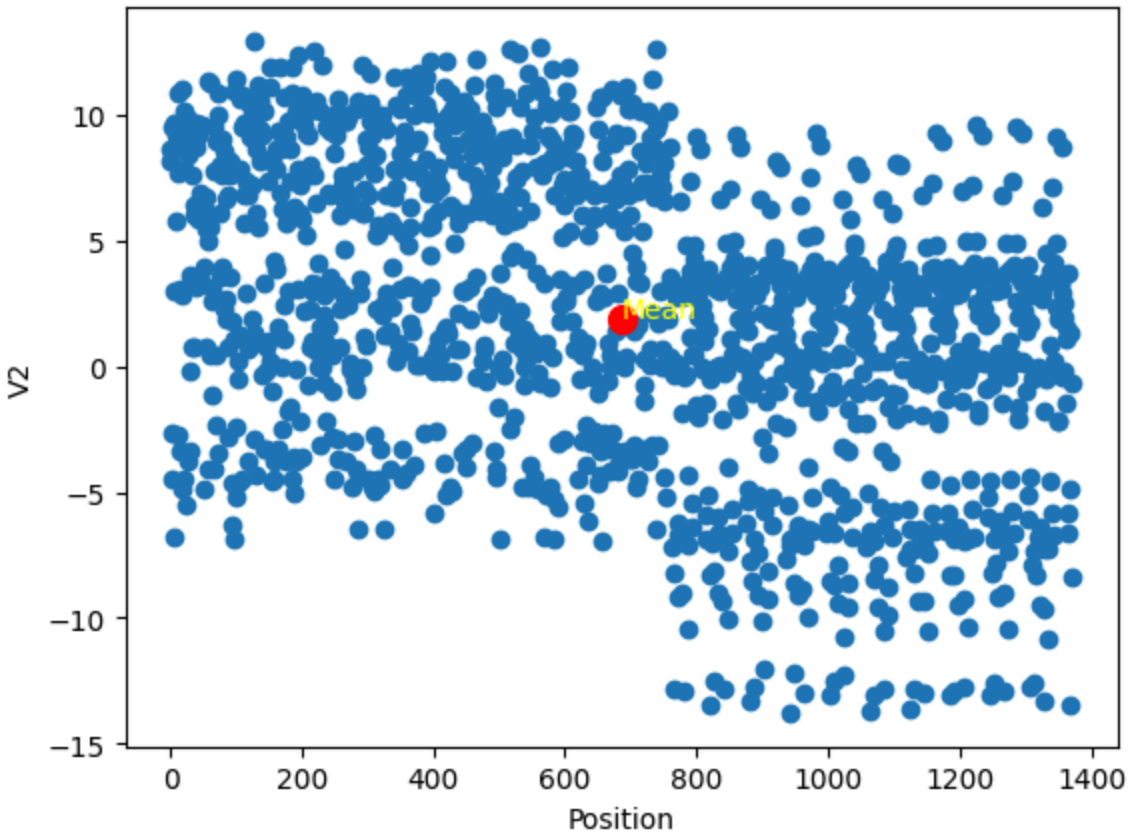
Out[77]:
```
Text(686.0, 0.43373525728862977, 'Mean')
```



- Visually V2

In [78]:
```
plt.xlabel("Position")
plt.ylabel("V2")
plt.scatter([i for i in range(rows)], dt['V2'])
plt.scatter(rows/2, means[1], color='red', s=100)
plt.text(rows/2, means[1], 'Mean', color='yellow')
```

Out[78]:
```
Text(686.0, 1.9223531209912539, 'Mean')
```



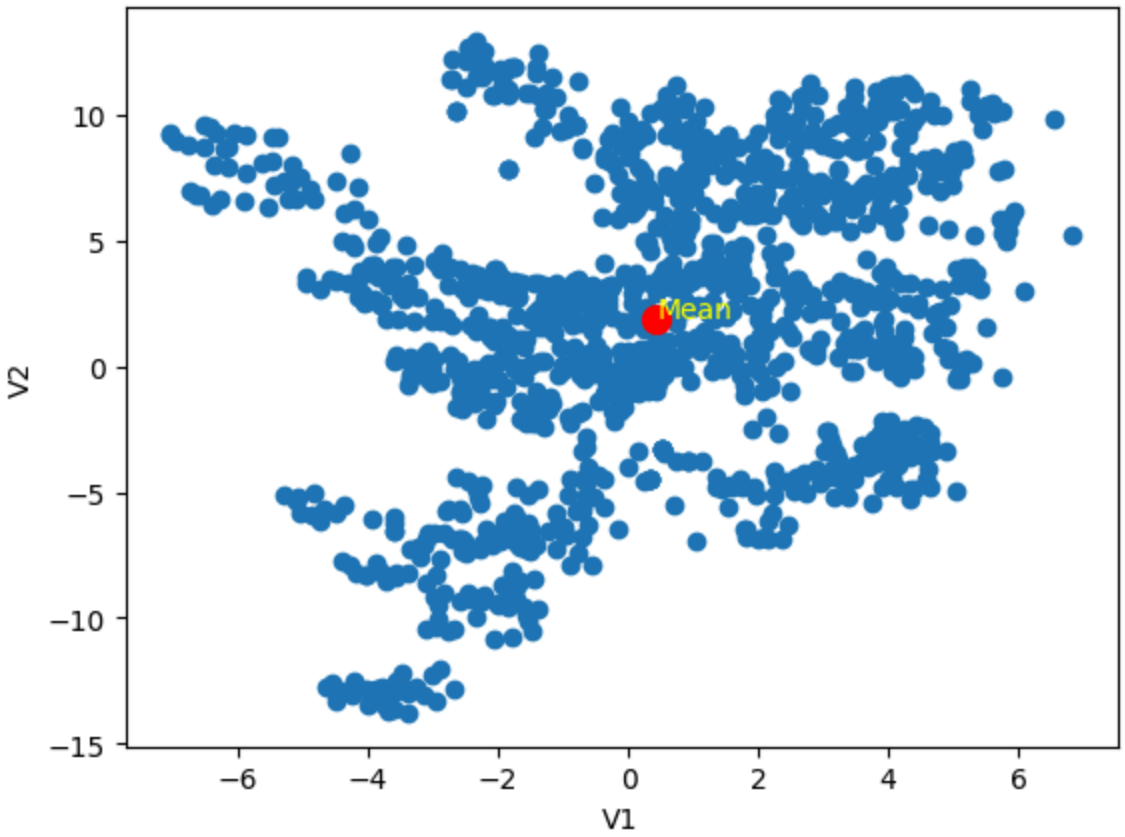
- V1 and V2 scatter plot

In [137…
```
plt.xlabel("V1")
plt.ylabel("V2")

plt.scatter(dt['V1'], dt['V2'])

plt.scatter(means[0], means[1], color='red', s=100)
plt.text(means[0], means[1], 'Mean', color='yellow')
```

Out[137… `Text(0.43373525728862977, 1.9223531209912539, 'Mean')`



The current plot doesn't provide a clear view of our dataset. There are numerous data points ranging from low to high within a single column. For instance, when V1 ≈ -4, the data points for V2 are scattered at different heights, making it difficult to understand the clustering.

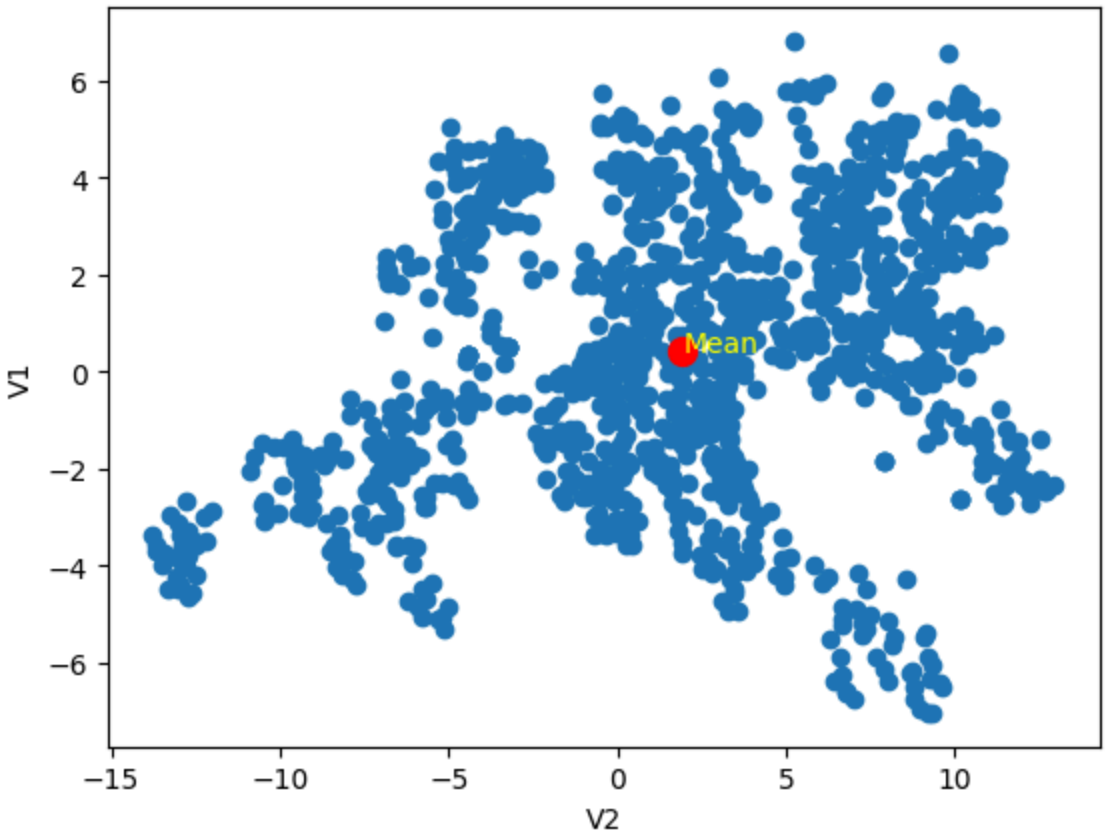
So instead, we can try to change the direction of the graph from *(V1, V2)* to *(V2, V1)*

In [138…
```
plt.xlabel("V2")
plt.ylabel("V1")

plt.scatter(dt['V2'], dt['V1'])

plt.scatter(means[1], means[0], color='red', s=100)
plt.text(means[1], means[0], 'Mean', color='yellow')
```

Out[138…
```
Text(1.9223531209912539, 0.43373525728862977, 'Mean')
```



"Now, we can see that it is much better. Additionally, we can easily see the four distinct clusters in the data. It will be:

- Low in *V2* and low in *V1*
- High in *V2* and low in *V1*
- Hgh in *V1* and low in *V2*
- High in *V1* and high in *V2*

Normalization of data

Min-Max normalization:

> - $$x' = \frac{\sum_{i=1}^{n}(x_{ij} - x_{min})}{x_{max} - x_{min}}$$

In [80]:
```
def normalization(arr, cols):
    norm=[] #Create empty restoring place
    for i in range(cols):
        print(f'Col #{i}:', end=' ')
        norm.append((arr[:,i]-arr[:,i].min())/(arr[:,i].max()-arr[:,i].min()))
        print(norm[i])
    return norm

def reshaped(arr):
    arr=np.array(arr)
```

```
    col=len(arr)
    row=len(arr[0])
    res=np.array([[None]*col]*row)
    for i in range(len(arr)):
        res[:,i]=arr[i]
    return res
```

The input array for K-Means should be a 2D array following this structure: [[*value 0 of col 1*, *value 1 of col 2*], ...,[*value n of col 1*, *value n of col 2*]]. We can see that dt.values already has the correct structure, which is [[*V1[0], V2[0]*],... **V1[n], V2[n]*]. So there's no need to reshape it.

In [130…
```
#Because, with my norm function it returned the array with shape [[values of col 1]
#With the reshaped function it will transform the data from [[a1, a2, ..., an], [b1
#to
# [[a1, b1],
#  [a2, b2],
#  ...
#  [an, bn]]
norm = reshaped(normalization(dt.values, cols))
```

```
Col #0: [0.76900389 0.83565902 0.78662859 ... 0.23738543 0.25084193 0.32452819]
Col #1: [0.83964273 0.82098209 0.41664827 ... 0.01176814 0.20170105 0.49074676]
```

In [131…
```
meansNorm=[]
stdsNorm=[]
for i in range(cols):
    vCol=norm[:,i]
    print(f'V{i+1} \n Range: {vCol.max()-vCol.min()}, Mean: {vCol.mean()}, Std: {vC
```

```
V1
 Range: 1.0, Mean: 0.5391136632764809, Std: 0.2049287443629144
V2
 Range: 1.0, Mean: 0.5873013774145724, Std: 0.21953127587108529
```

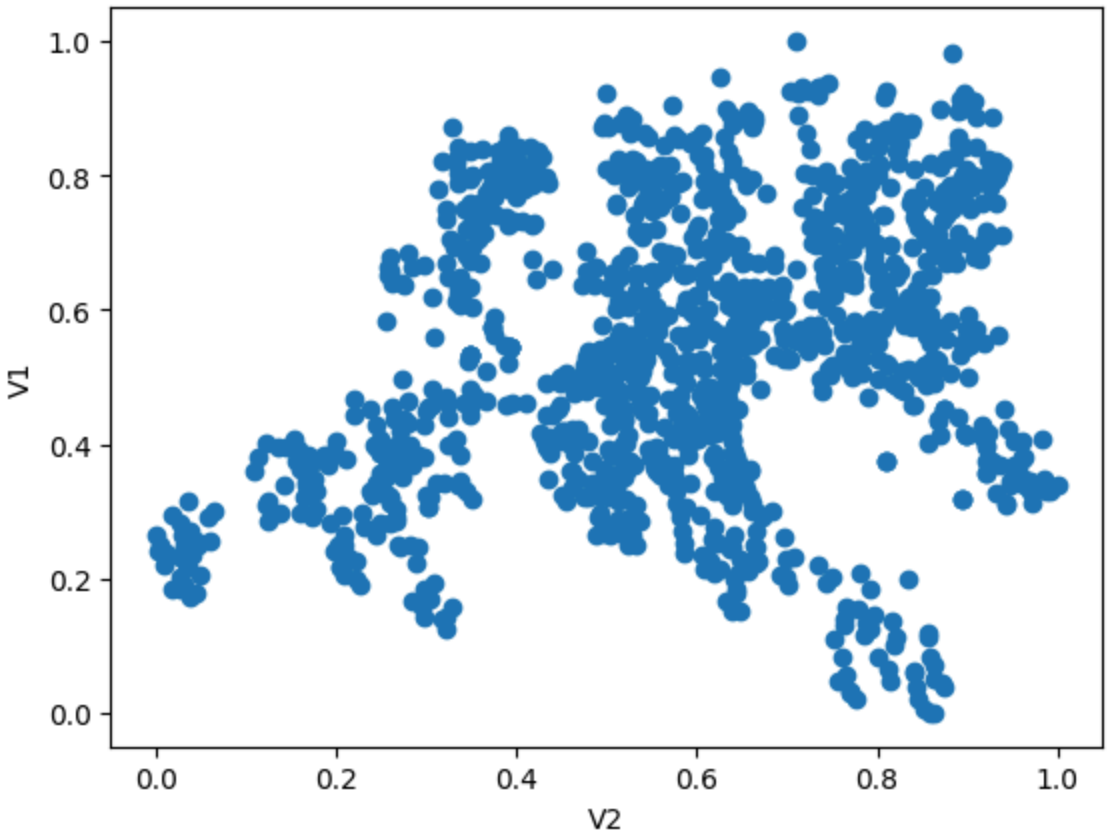
Now, we can see that the *ranges, means, and standard deviations* of the two columns are approximately the same! A greate improvement!

- Visualize V1 and V2 scatter plot after normalization

In [132…
```
plt.xlabel("V2")
plt.ylabel("V1")

plt.scatter(norm[:,1], norm[:,0])
```

Out[132…
```
<matplotlib.collections.PathCollection at 0x1e9b4dc3790>
```



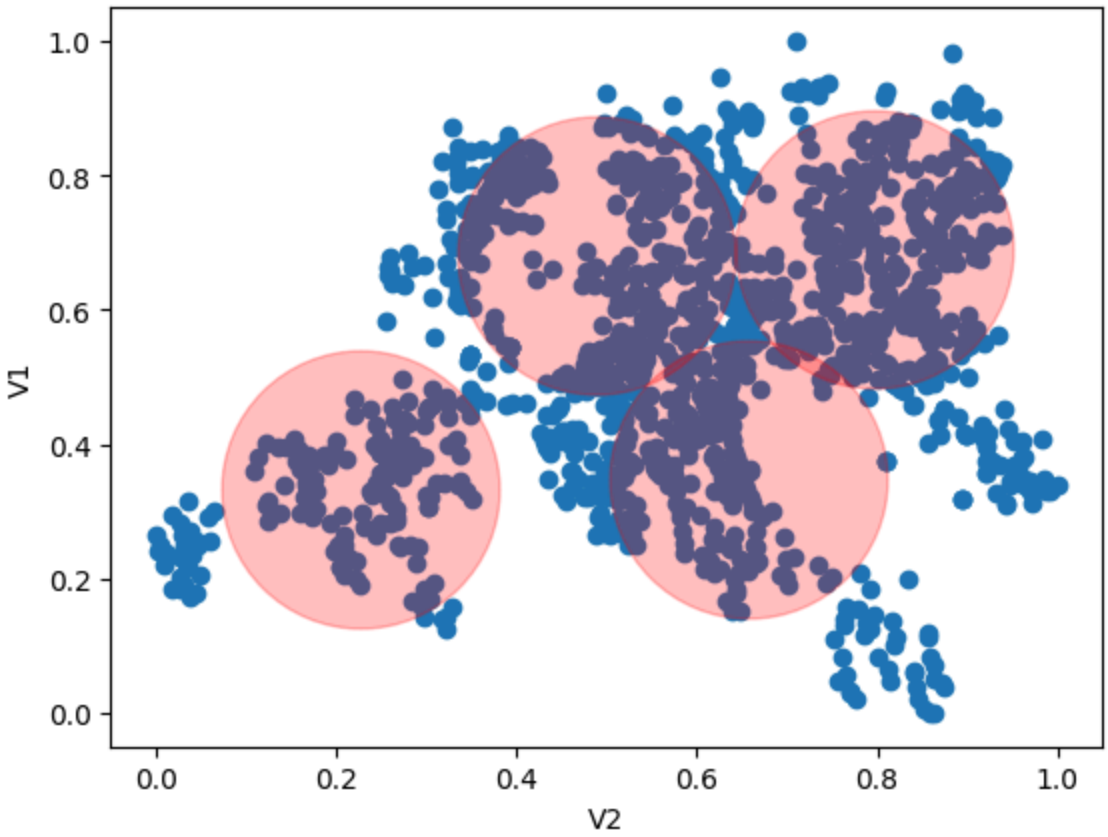
In [133…
```
#Apply K-means clustering to the normalized data
km_res=KMeans(n_clusters=4).fit(norm)
```

In here, we are getting a greate result with 4 clusters.

In [135…
```
plt.xlabel("V2")
plt.ylabel("V1")

plt.scatter(norm[:,1], norm[:,0])
plt.scatter(km_res.cluster_centers_[:,1], km_res.cluster_centers_[:,0], color='red'
```

Out[135…
```
<matplotlib.collections.PathCollection at 0x1e9b4ef77d0>
```



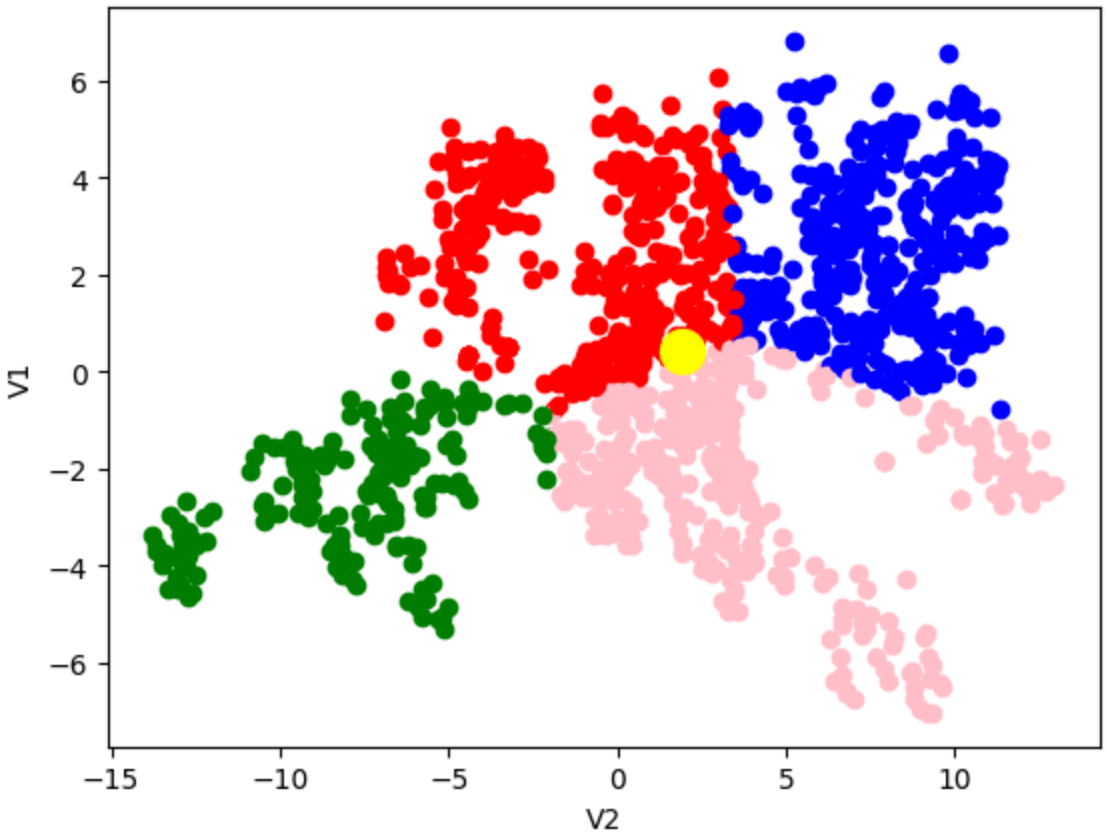
From here, I will mark on 4 areas belonging to each cluster. The color represents the cluster label. The yellow point represents the mean of the entire dataset.

In [139…
```
labels=km_res.labels_ #Getting the labels of each data point
```

In [140…
```
plt.xlabel('V2')
plt.ylabel('V1')
colors=['red','blue','green','pink']
# Loop through each cluster
for i in range(rows):
    plt.scatter(dt.iloc[i][1], dt.iloc[i][0], color=colors[labels[i]])
plt.scatter(means[1],means[0], color='Yellow', s=250)
```

```
C:\Users\DELL\AppData\Local\Temp\ipykernel_28780\1651953459.py:6: FutureWarning: Ser
ies.__getitem__ treating keys as positions is deprecated. In a future version, integ
er keys will always be treated as labels (consistent with DataFrame behavior). To ac
cess a value by position, use `ser.iloc[pos]`
  plt.scatter(dt.iloc[i][1], dt.iloc[i][0], color=colors[labels[i]])
```

Out[140…
```
<matplotlib.collections.PathCollection at 0x1e99cd36450>
```



We can see that K-means clustering has correctly classified the data points into four distinct clusters.

- The yellow point represents the mean of the entire dataset.
- Green points belong to the cluster with the lowest *V2* and lowest *V1*.
- Red points belong to the cluster with the highest *V1* and low *V2*.
- Blue points belong to the cluster with the highest *V1* and highest *V2*.
- Pink points belong to the cluster with the lowest *V1* and highest *V2*.

This result shows that ***K-means clustering*** is a powerful and effective technique for clustering data as well as ***the normalization*** step significantly improved the clustering results, making it easier to interpret and visualize the data. So, to ephasize the important of normalization in data preprocessing, I will also illustrate the result of **K-means clustering without normalization** below.

In [141…
```
km_res_old=KMeans(n_clusters=4).fit(dt) #Taking dt as a raw input without normaliza
```

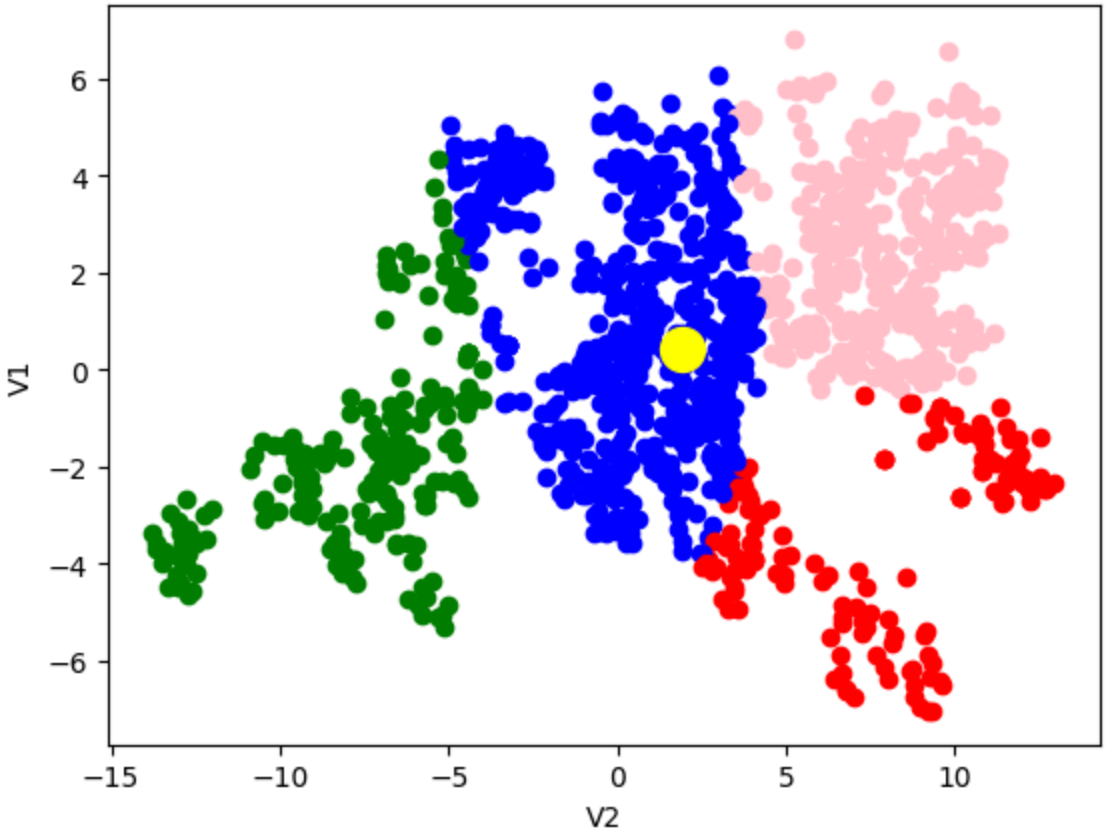
In [142…
```
labels=km_res_old.labels_
```

In [143…
```
plt.xlabel('V2')
plt.ylabel('V1')
colors=['red','blue','green','pink']
# Loop through each cluster
for i in range(rows):
```

```
    plt.scatter(dt.iloc[i][1], dt.iloc[i][0], color=colors[labels[i]])
plt.scatter(means[1],means[0], color='Yellow', s=250)
```

```
C:\Users\DELL\AppData\Local\Temp\ipykernel_28780\1651953459.py:6: FutureWarning: Ser
ies.__getitem__ treating keys as positions is deprecated. In a future version, integ
er keys will always be treated as labels (consistent with DataFrame behavior). To ac
cess a value by position, use `ser.iloc[pos]`
  plt.scatter(dt.iloc[i][1], dt.iloc[i][0], color=colors[labels[i]])
```

Out[143… `<matplotlib.collections.PathCollection at 0x1e9b923c990>`



We can see that K-means clustering without normalization has also correctly classified the data points into four distinct clusters. However, without normalization, the clusters are skewed, likely because one feature (V1 or V2) has a larger range of values than the other. The lack of normalization can lead to the algorithm being biased towards features with larger ranges. The clusters seem to overlap more, which could reduce the accuracy of the model.

In conclusion, the normalization step is crucial for data preprocessing in K-means clustering. By normalizing the data, we ensure that all features are on the same scale, which can help the algorithm converge!