18bce101

Data mining practical 2

Aim: Apply Data Smoothing using Binning Methods:

1) Smoothing using Bin Mean

2) By Bin Median

3) bin boundary smoothing

Data binning is a technique used in data processing and statistics. Binning method is used to smoothing data or to handle noisy data. Binning is a form of quantization (larger set to smaller set). Binning is also used in image processing.

Bin means: Each value stored in the bin will be replaced by bin means.

Bin median: Each value stored in the bin will be replaced by bin median.

Bin boundary: The minimum and maximum bin values are stored at the boundary while intermediate bin values are replaced by the boundary value to which it is more closer.

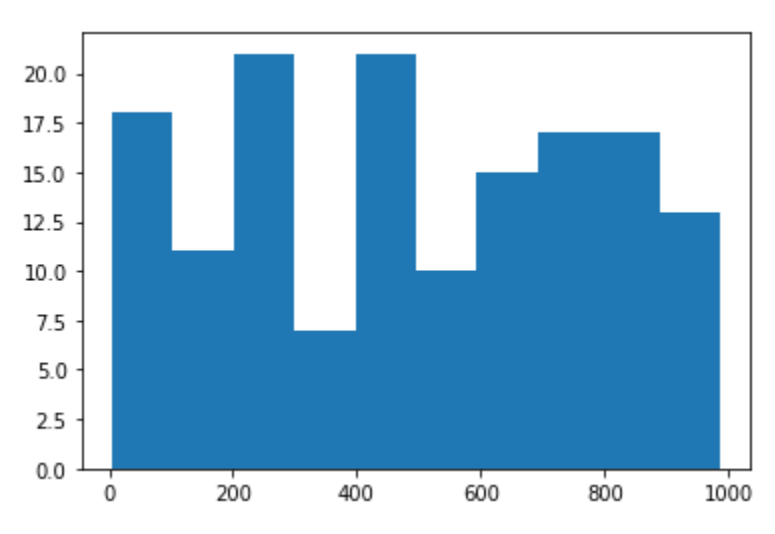
In [2]:
```
#importing necessary libraries

from matplotlib import pyplot as plt
import pandas as pd
import numpy as np
import seaborn as sns
```

Working on a sample data

In [3]:
```
#generating an array of numbers and sorting them for binning
data = np.random.randint(1,1000,150)
data = np.sort(data)
print(data)

plt.hist(data)
```

```
[  4  10  21  29  44  52  55  60  67  71  76  85  87  87  88  89  94  98
 104 109 121 140 142 142 165 186 191 199 200 202 208 214 221 238 240 242
 244 246 248 251 253 258 259 275 278 289 290 290 292 294 302 317 343 356
 365 378 380 402 408 411 413 414 415 422 424 424 425 427 435 437 438 443
 444 465 471 479 491 493 505 511 511 526 539 562 569 588 590 593 595 606
 606 622 624 626 626 629 646 649 663 668 683 687 689 700 705 732 737 737
 738 740 747 756 765 767 769 773 777 780 783 789 792 796 804 808 813 815
 818 825 832 840 842 844 853 858 862 863 868 891 897 904 906 918 919 921
 923 948 954 965 984 987]
```

Out[3]:
```
(array([18., 11., 21.,  7., 21., 10., 15., 17., 17., 13.]),
 array([  4. , 102.3, 200.6, 298.9, 397.2, 495.5, 593.8, 692.1, 790.4,
        888.7, 987. ]),
 <a list of 10 Patch objects>)
```



In [4]:
```
#making 3 bins
bin1 = np.zeros((5,30))
bin2 = np.zeros((5,30))
bin3 = np.zeros((5,30))

binsize = len(data) // 5
bins = np.zeros((5,30))

for i in range(0, len(data), binsize):
    k = int(i/binsize)
    for j in range(binsize):
        bins[k,j] = data[i+j]

print(bins)
```

```
[[  4.  10.  21.  29.  44.  52.  55.  60.  67.  71.  76.  85.  87.  87.
   88.  89.  94.  98. 104. 109. 121. 140. 142. 142. 165. 186. 191. 199.
  200. 202.]
 [208. 214. 221. 238. 240. 242. 244. 246. 248. 251. 253. 258. 259. 275.
  278. 289. 290. 290. 292. 294. 302. 317. 343. 356. 365. 378. 380. 402.
  408. 411.]
 [413. 414. 415. 422. 424. 424. 425. 427. 435. 437. 438. 443. 444. 465.
  471. 479. 491. 493. 505. 511. 511. 526. 539. 562. 569. 588. 590. 593.
  595. 606.]
 [606. 622. 624. 626. 626. 629. 646. 649. 663. 668. 683. 687. 689. 700.
  705. 732. 737. 737. 738. 740. 747. 756. 765. 767. 769. 773. 777. 780.
  783. 789.]
 [792. 796. 804. 808. 813. 815. 818. 825. 832. 840. 842. 844. 853. 858.
  862. 863. 868. 891. 897. 904. 906. 918. 919. 921. 923. 948. 954. 965.
  984. 987.]]
```

In [5]:
```
#bin mean

for i in range(bins.shape[0]):
    sum = 0;
    for j in range(bins.shape[1]):
        sum += bins[i,j]
    mean = sum/30
    for j in range(bins.shape[1]):
        bin1[i,j] = mean

print(bin1)
```

```
[[100.6        100.6        100.6        100.6        100.6
  100.6        100.6        100.6        100.6        100.6
  100.6        100.6        100.6        100.6        100.6
  100.6        100.6        100.6        100.6        100.6
  100.6        100.6        100.6        100.6        100.6
  100.6        100.6        100.6        100.6        100.6       ]
 [293.06666667 293.06666667 293.06666667 293.06666667 293.06666667
  293.06666667 293.06666667 293.06666667 293.06666667 293.06666667
  293.06666667 293.06666667 293.06666667 293.06666667 293.06666667
  293.06666667 293.06666667 293.06666667 293.06666667 293.06666667
  293.06666667 293.06666667 293.06666667 293.06666667 293.06666667
  293.06666667 293.06666667 293.06666667 293.06666667 293.06666667]
 [488.5        488.5        488.5        488.5        488.5
  488.5        488.5        488.5        488.5        488.5
  488.5        488.5        488.5        488.5        488.5
  488.5        488.5        488.5        488.5        488.5
  488.5        488.5        488.5        488.5        488.5
  488.5        488.5        488.5        488.5        488.5       ]
 [707.1        707.1        707.1        707.1        707.1
  707.1        707.1        707.1        707.1        707.1
  707.1        707.1        707.1        707.1        707.1
  707.1        707.1        707.1        707.1        707.1
  707.1        707.1        707.1        707.1        707.1
  707.1        707.1        707.1        707.1        707.1       ]
 [875.         875.         875.         875.         875.
  875.         875.         875.         875.         875.
  875.         875.         875.         875.         875.
  875.         875.         875.         875.         875.
  875.         875.         875.         875.         875.
  875.         875.         875.         875.         875.        ]]
```

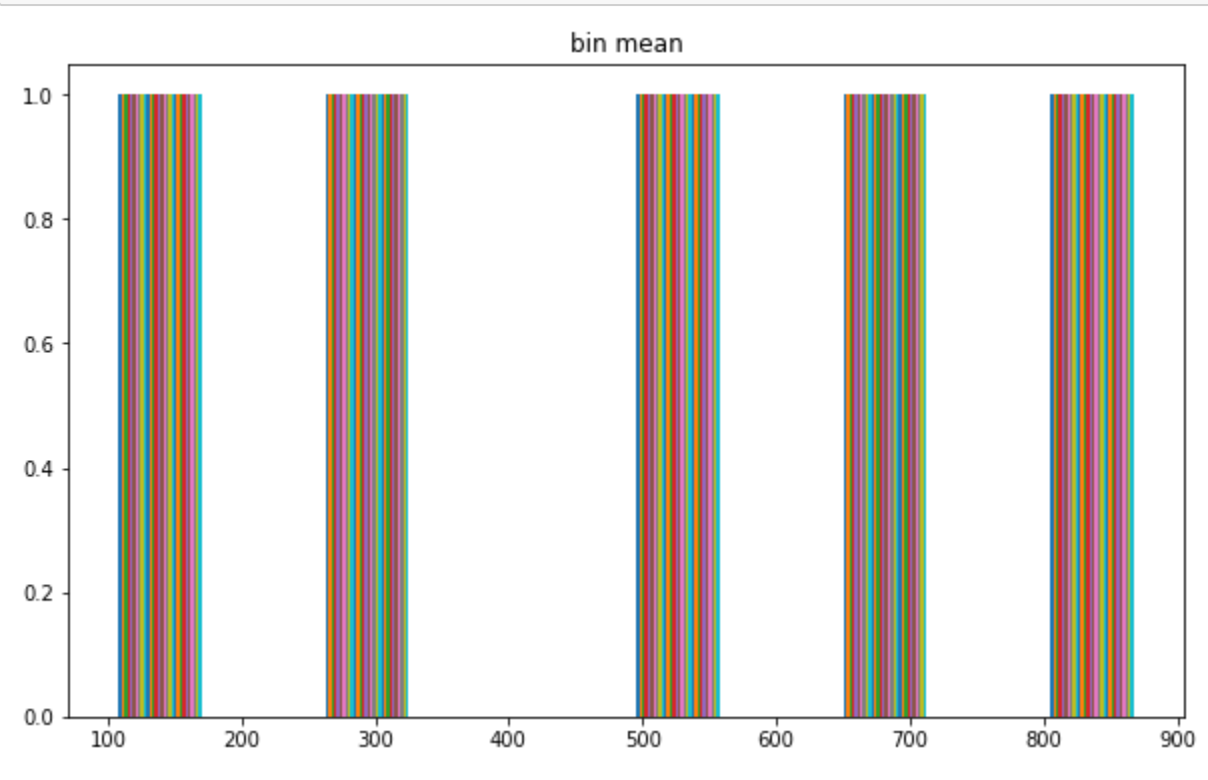
In [6]:
```
#bin median

for i in range(bins.shape[0]):
    median = bins[i,i+15]
    for j in range(bins.shape[1]):
        bin2[i,j] = median

print(bin2)
```

```
[[ 89.  89.  89.  89.  89.  89.  89.  89.  89.  89.  89.  89.  89.  89.
   89.  89.  89.  89.  89.  89.  89.  89.  89.  89.  89.  89.  89.  89.
   89.  89.]
 [290. 290. 290. 290. 290. 290. 290. 290. 290. 290. 290. 290. 290. 290.
  290. 290. 290. 290. 290. 290. 290. 290. 290. 290. 290. 290. 290. 290.
  290. 290.]
 [493. 493. 493. 493. 493. 493. 493. 493. 493. 493. 493. 493. 493. 493.
  493. 493. 493. 493. 493. 493. 493. 493. 493. 493. 493. 493. 493. 493.
  493. 493.]
 [738. 738. 738. 738. 738. 738. 738. 738. 738. 738. 738. 738. 738. 738.
  738. 738. 738. 738. 738. 738. 738. 738. 738. 738. 738. 738. 738. 738.
  738. 738.]
 [904. 904. 904. 904. 904. 904. 904. 904. 904. 904. 904. 904. 904. 904.
  904. 904. 904. 904. 904. 904. 904. 904. 904. 904. 904. 904. 904. 904.
  904. 904.]]
```

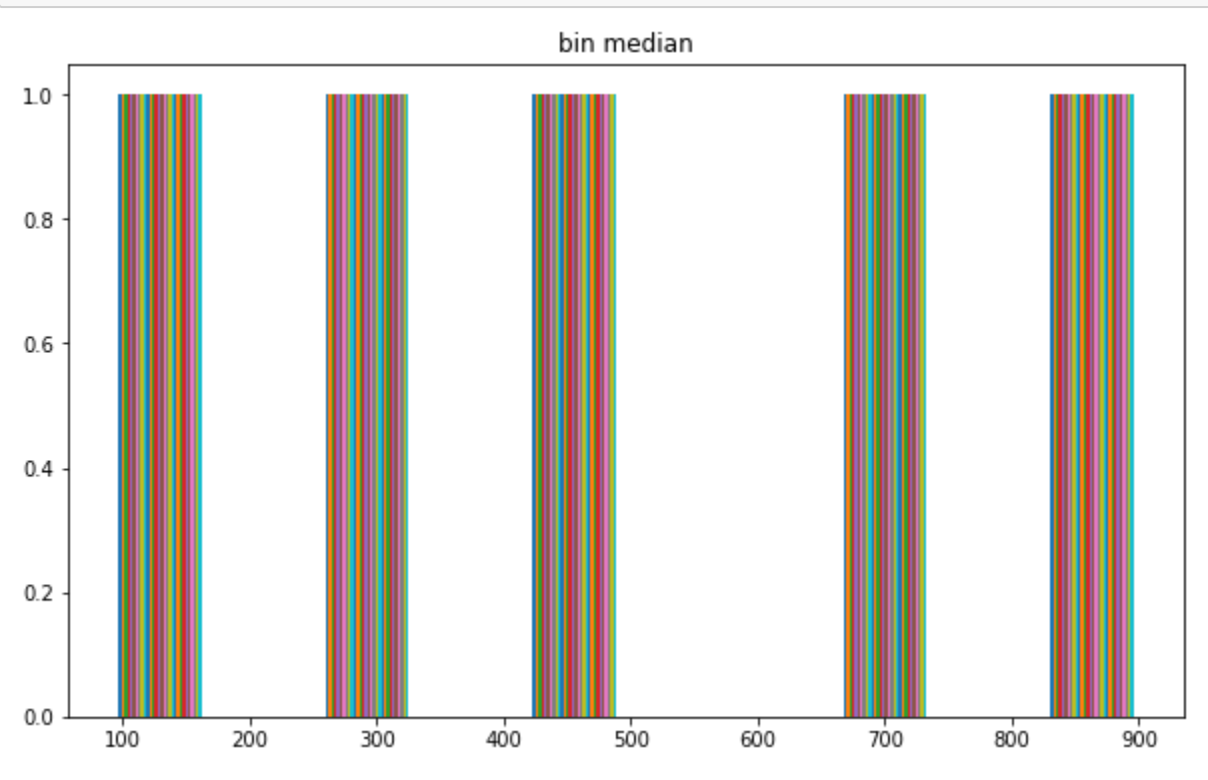
In [7]:
```
#bin boundary

for i in range(bins.shape[0]):
    left = bins[i,0]
    right = bins[i,29]
    for j in range(bins.shape[1]):
        bin3[i,j] = min(bins[i,j] - left, right - bins[i,j])

print(bin3)
```

```
[[ 0.  6. 17. 25. 40. 48. 51. 56. 63. 67. 72. 81. 83. 83. 84. 85. 90. 94.
  98. 93. 81. 62. 60. 60. 37. 16. 11.  3.  2.  0.]
 [ 0.  6. 13. 30. 32. 34. 36. 38. 40. 43. 45. 50. 51. 67. 70. 81. 82. 82.
  84. 86. 94. 94. 68. 55. 46. 33. 31.  9.  3.  0.]
 [ 0.  1.  2.  9. 11. 11. 12. 14. 22. 24. 25. 30. 31. 52. 58. 66. 78. 80.
  92. 95. 95. 80. 67. 44. 37. 18. 16. 13. 11.  0.]
 [ 0. 16. 18. 20. 20. 23. 40. 43. 57. 62. 77. 81. 83. 89. 84. 57. 52. 52.
  51. 49. 42. 33. 24. 22. 20. 16. 12.  9.  6.  0.]
 [ 0.  4. 12. 16. 21. 23. 26. 33. 40. 48. 50. 52. 61. 66. 70. 71. 76. 96.
  90. 83. 81. 69. 68. 66. 64. 39. 33. 22.  3.  0.]]
```

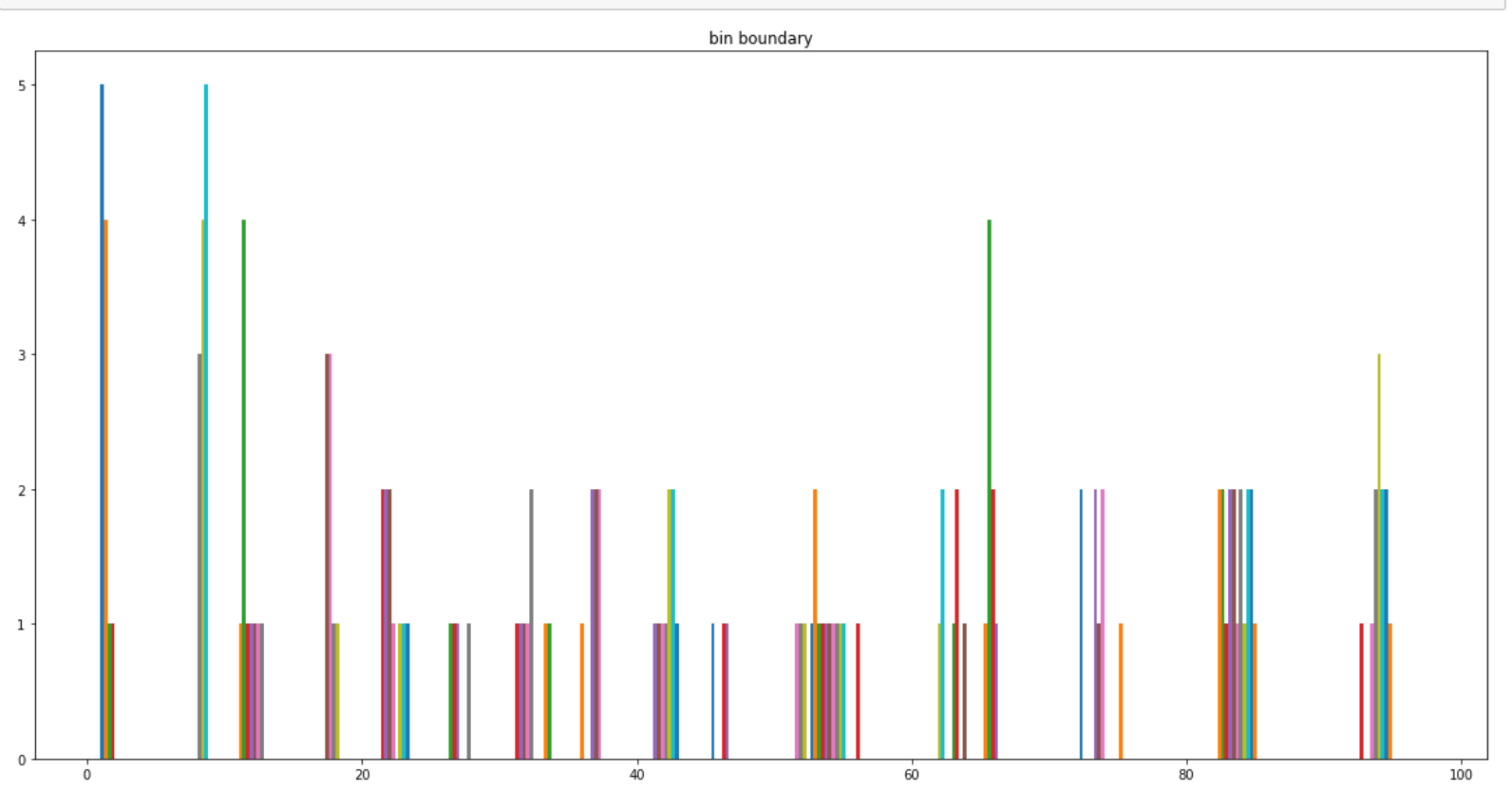
In [8]:
```
#visualization of all the three methods
fig= plt.figure(figsize=(10,6))
plt.title("bin mean")
plt.hist(bin1)
plt.show()
```



In [9]:
```
fig= plt.figure(figsize=(10,6))
plt.title("bin median")
plt.hist(bin2)
plt.show()
```



In [10]:
```
fig= plt.figure(figsize=(20,10))
plt.title("bin boundary")
plt.hist(bin3)
plt.show()
```



Real world example

Specific strategies of binning data include fixed-width and adaptive binning.

This dataset is used to predict whether a patient is likely to get stroke based on the input parameters like gender, age, and various diseases and smoking status.

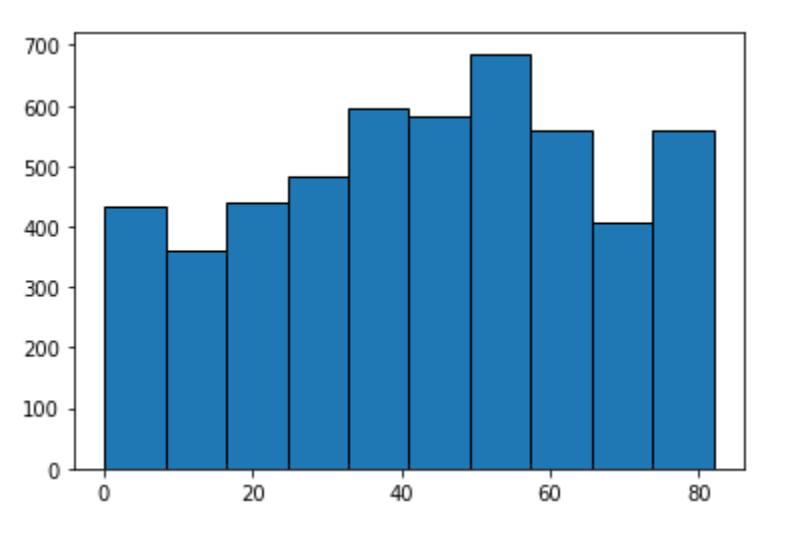
In [13]:
```
datadf = pd.read_csv('healthcare-dataset-stroke-data.csv')
datadf.head()
```

Out[13]:

	id	gender	age	hypertension	heart_disease	ever_married	work_type	Residence_type	avg_glucose_level	bmi	smol
0	9046	Male	67.0	0	1	Yes	Private	Urban	228.69	36.6	form
1	51676	Female	61.0	0	0	Yes	Self-employed	Rural	202.21	NaN	ne
2	31112	Male	80.0	0	1	Yes	Private	Rural	105.92	32.5	ne
3	60182	Female	49.0	0	0	Yes	Private	Urban	171.23	34.4	
4	1665	Female	79.0	1	0	Yes	Self-employed	Rural	174.12	24.0	ne

In [37]:
```
plt.hist(datadf['age'], edgecolor='black')
```

Out[37]:
```
(array([434., 362., 440., 484., 597., 583., 686., 559., 407., 558.]),
 array([8.0000e-02, 8.2720e+00, 1.6464e+01, 2.4656e+01, 3.2848e+01,
        4.1040e+01, 4.9232e+01, 5.7424e+01, 6.5616e+01, 7.3808e+01,
        8.2000e+01]),
 <a list of 10 Patch objects>)
```



In [15]:
```
datadf['age'].describe()
```

Out[15]:
```
count    5110.000000
mean       43.226614
std        22.612647
min         0.080000
25%        25.000000
50%        45.000000
75%        61.000000
max        82.000000
Name: age, dtype: float64
```

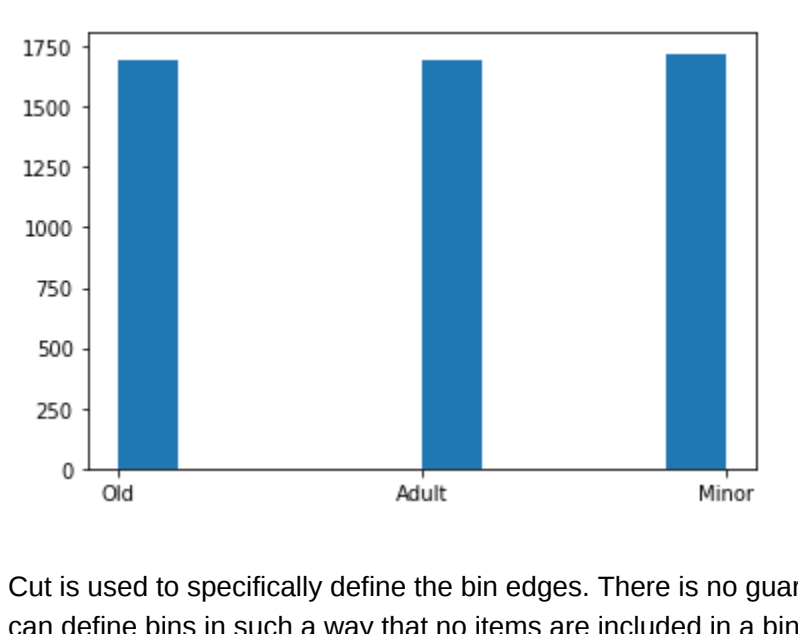
This function tries to divide the data into equal-sized bins. The bins are defined using percentiles, based on the distribution and not on the actual numeric edges of the bins.

In [38]:
```
labels= ['Minor' , 'Adult' , 'Old']
datadf['Age_Category']=pd.qcut(datadf['age'],q=3,labels=labels , precision=0)
pd.qcut(datadf['age'],q=3, precision=0).value_counts()
```

Out[38]:
```
(-0.9, 32.0]    1720
(55.0, 82.0]    1696
(32.0, 55.0]    1694
Name: age, dtype: int64
```

In [39]:
```
plt.hist(datadf['Age_Category'])
```

Out[39]:
```
(array([1696.,    0.,    0.,    0.,    0., 1694.,    0.,    0.,    0.,
        1720.]),
 array([0. , 0.2, 0.4, 0.6, 0.8, 1. , 1.2, 1.4, 1.6, 1.8, 2. ]),
 <a list of 10 Patch objects>)
```



Cut is used to specifically define the bin edges. There is no guarantee about the distribution of items in each bin. In fact, you can define bins in such a way that no items are included in a bin or nearly all items are in a single bin.

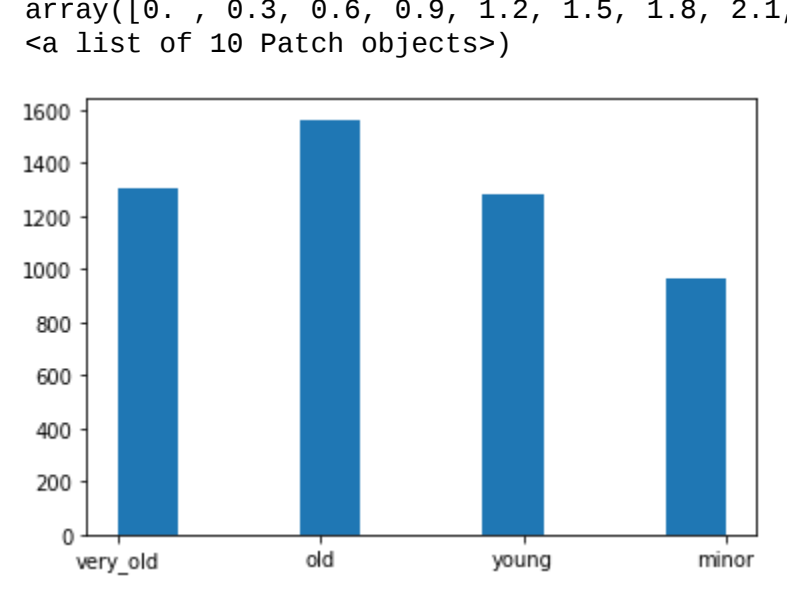
In [34]:
```
minimum = datadf['age'].min()
maximum = datadf['age'].max()
print('Minimum Age: ', minimum)
print('Maximum Age: ', maximum)

bins = [minimum] + [19] + [40] + [60] + [maximum]
labels = ['minor','young','old', 'very_old']

datadf['age_range'] = pd.cut(datadf['age'],bins = bins, labels = labels, include_lowest=True
)
pd.cut(datadf['age'],bins = bins, labels = labels, include_lowest=True).value_counts()
```

```
Minimum Age:  0.08
Maximum Age:  82.0
```

Out[34]:
```
old         1562
very_old    1304
young       1278
minor        966
Name: age, dtype: int64
```

In [35]:
```
plt.hist(datadf['age_range'])
```

Out[35]:
```
(array([1304.,    0.,    0., 1562.,    0.,    0., 1278.,    0.,    0.,
         966.]),
 array([0. , 0.3, 0.6, 0.9, 1.2, 1.5, 1.8, 2.1, 2.4, 2.7, 3. ]),
 <a list of 10 Patch objects>)
```



Conclusion

The method used for the Stroke prediction dataset is *adaptive binning*. Since qcut distributes in equal ranges, it is not much of a use for us, instead the method cut wherein we distribute according to our bin range is appropiate for our dataset for furthur analysis and prediction. Another common example of binning is Pixel binning where we bin the pixels of image.