HW2

February 11, 2019

0.1 Homework 2

Erin Witmer
CS 440

0.1.1 Chapter 2

[2.3] Suppose that the values for a given set of data are grouped into intervals. The intervals and corresponding frequencies are as follows. Compute an approximate median value.

In [57]:
```
# calculate n, the total observations
# n/2 is the median value
import numpy as np
f = [200, 450, 300, 1500, 700, 44]
n = np.sum(f)
median_value = n/2
median_value
```

Out[57]: 1597.0

In [58]:
```
# calculate the cumulative observations to determine the bin the median falls in
tot = 0
for i in f:
    tot += i
    print(tot)
```

```
200
650
950
2450
3150
3194
```

In [59]:
```
# the median value falls in the bin 21-50
bin_min = 21
bin_max = 50
bin_range = bin_max - bin_min
bin_range
```

```
Out[59]: 29

In [60]: # calculate how far into the bin the median value falls
         n_remaining = median_value-(f[2]+f[1]+f[0])
         bin_size = f[3]
         pct_bin = n_remaining / bin_size
         pct_bin

Out[60]: 0.43133333333333335

In [61]: # make an assumption that the bin is evenly distributed
         # the median value falls 43% into the bin
         est_median = bin_range * pct_bin + bin_min
         print(round(est_median,2))

33.51
```

Making an assumption that the bin in which the median value falls is evenly distributed, the median value is approximately 33.5.

[2.4] Suppose that a hospital tested the age and body far for 18 randomly selected adults with the following results.

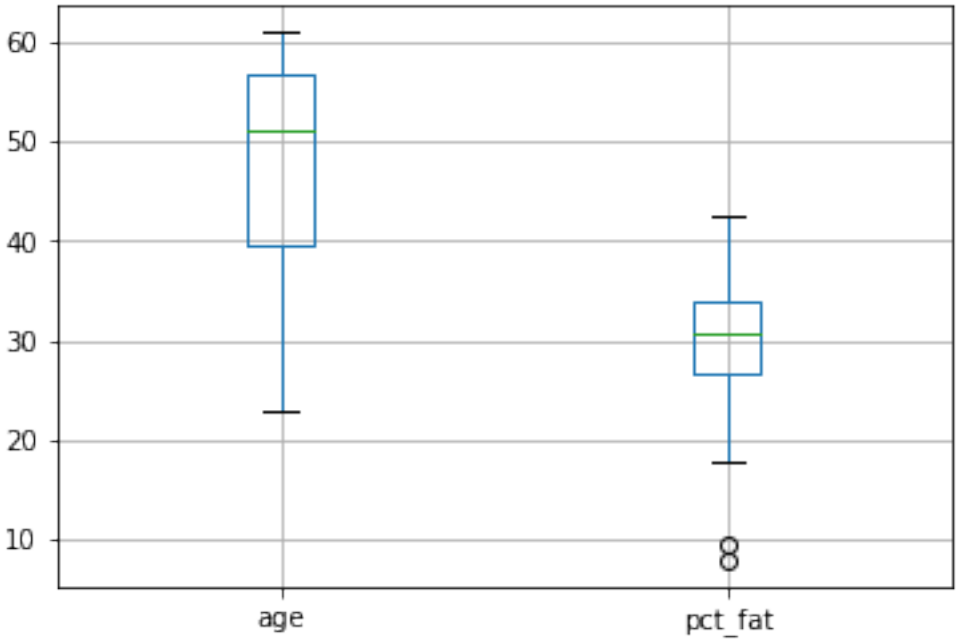
(a) Calculate the mean, median and std.dev of age and % fat

```
In [62]: import pandas as pd
         age = [23,23,27,27,39,41,47,49,50,52,54,54,56,57,58,58,60,61]
         pct_fat = [9.5,26.5,7.8,17.8,31.4,25.9,27.4,27.2,31.2,34.6,
                    42.5,28.8,33.4,30.2,34.1,32.9,41.2,35.7]
         df = pd.DataFrame({'age': age, 'pct_fat': pct_fat})
         df.describe()

Out[62]:              age    pct_fat
         count  18.000000  18.000000
         mean   46.444444  28.783333
         std    13.218624   9.254395
         min    23.000000   7.800000
         25%    39.500000  26.675000
         50%    51.000000  30.700000
         75%    56.750000  33.925000
         max    61.000000  42.500000

In [63]: df.std(ddof=0)

Out[63]: age        12.846194
         pct_fat     8.993655
         dtype: float64
```

Age: mean = 46.4, median = 51, std (corrected to n-1 since this is a sample) = 13.2, std (uncorrected) = 12.8
%Fat: mean = 28.8, median = 30.7, std (corrected to n-1 since this is a sample) = 9.3, std (uncorrected) = 9.0

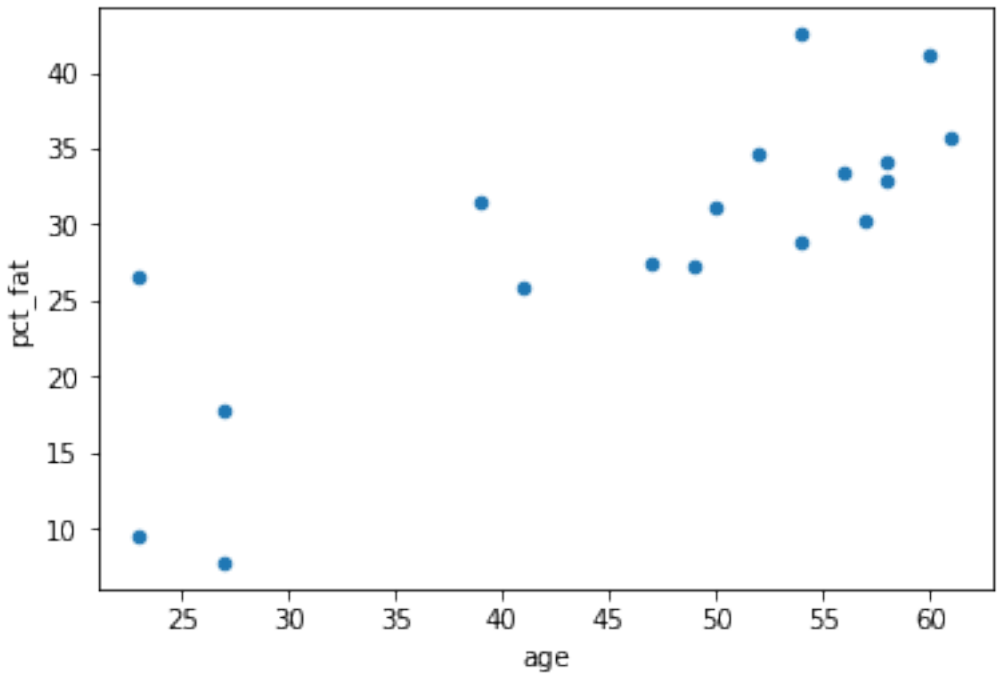
(b) Draw boxplots for age and %fat

```
In [64]: df.boxplot()

Out[64]: <matplotlib.axes._subplots.AxesSubplot at 0x11c09d710>
```

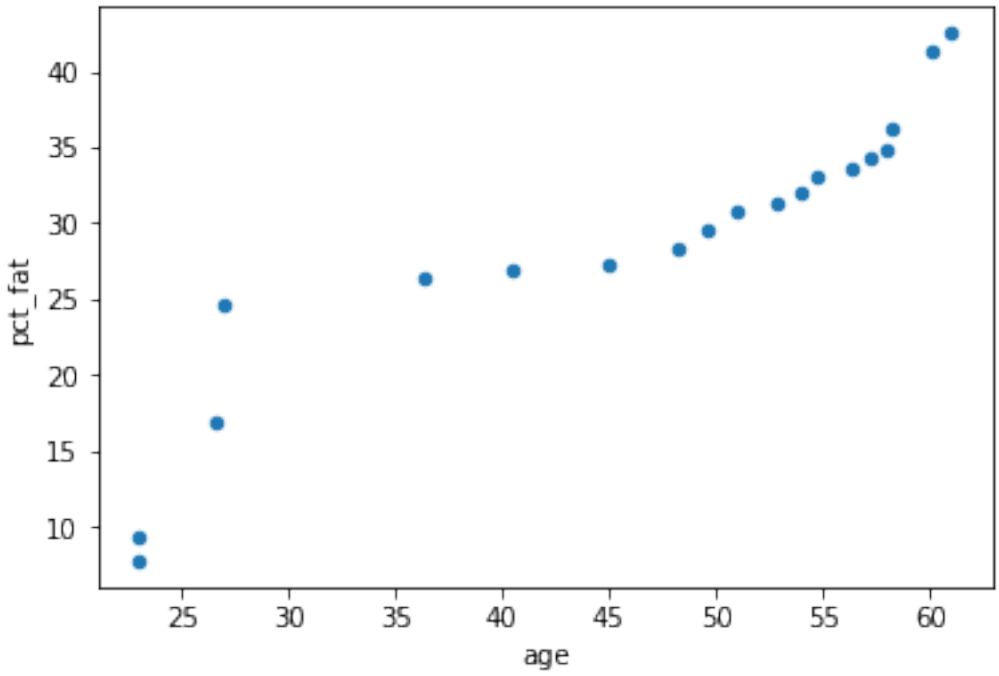


(c) Draw a scatter plot and quantile-quantile plot based on these two variables.

```
In [65]: df.plot.scatter(x='age',y='pct_fat')

Out[65]: <matplotlib.axes._subplots.AxesSubplot at 0x1208c8f98>
```




```
In [66]: df_q = df.quantile(np.linspace(0, 1.0, num=19))

In [67]: df_q.plot.scatter(x='age',y='pct_fat')

Out[67]: <matplotlib.axes._subplots.AxesSubplot at 0x1209906a0>
```



[2.6] Given two objects represented by the tuples (22,1,42,10) and (20,0,36,8)
(a) Calculate the Euclidean distance between the two objects

```
In [68]: # generalized formula for distance in L norm
         def dist(obj1, obj2, l):
             total = 0
             for i in range(len(obj1)):
                 total += abs(obj1[i]-obj2[i]) ** l
             return total ** (1/l)

In [69]: # objects
         obj1 = [22,1,42,10]
         obj2 = [20, 0, 36, 8]

         #Euclidean (L2)
         dist(obj1, obj2, 2)

Out[69]: 6.708203932499369
```

The Euclidean distance is 6.7
(b) Calculate the Manhattan distance between the two objects

```
In [70]: #Manhattan (L1)
         dist(obj1, obj2, 1)

Out[70]: 11.0
```

The Manhattan distance is 11.0
(c) Calculate the Minkowski distance between the two objects (q=3)

```
In [71]: #Minkowski (L3)
         dist(obj1, obj2, 3)

Out[71]: 6.153449493663682
```

The Minkowski distance (L=3) is 6.2
(d) Calculate the supremum distance between the two objects

```
In [72]: # formula for supremum
         def supremum(obj1, obj2):
             diff = map(lambda x, y: abs(x - y), obj1, obj2)
             return np.max(list(diff))

In [73]: # Supremum
         supremum(obj1, obj2)

Out[73]: 6
```

The supremum distance is 6.

[2.8] It is important to define or select similarity measures in data analysis. However, there is no commonly accepted subjective simliarity measure. Results can vary based on the similarity measure used. Nonetheless, seemingly different similarity measures may be equivilent after some transformation. Suppose we have the following 2-D data set:

(a) Consider the data as 2-D data points. Given the new data point, x=(1.4,1.6) as a query, rank the database points based on similarity with the query using Euclidean distance, Manhattan distance, supremum distance, and cosine similarity

```
In [74]: # data points for comparison, new data point and indices
         x_all = [(1.5,1.7),(2.0,1.9),(1.6,1.8),(1.2,1.5),(1.5,1.0)]
         x_new = (1.4,1.6)
         ind = ['x1','x2','x3','x4','x5']

In [75]: # maps list into euclidean distance between each data point and the new data point
         eucl = list(map(lambda x: dist(x,x_new,2),x_all))

In [76]: # maps list into manhattan distance between each data point and the new data point
         nyc = list(map(lambda x: dist(x,x_new,1),x_all))

In [77]: # maps list into supremum distance between each data point and the new data point
         sup = list(map(lambda x: supremum(x,x_new),x_all))

In [78]: # formula to calculate cosine similarity
         from numpy import linalg as LA
         def cos_sim(obj1, obj2):
             d = np.dot(obj1, obj2) / (LA.norm(obj1) * LA.norm(obj2))
             return d

In [79]: # maps list into cosine similarity between each data point and the new data point
         cs = list(map(lambda x: cos_sim(x,x_new),x_all))

In [80]: # puts each of the mappings into a dataframe
         df = pd.DataFrame({'euclidean': eucl,
                            'manhattan': nyc,
                            'supremum': sup,
                            'cos_sim': cs},index=ind)
         df
```

Out[80]:

	euclidean	manhattan	supremum	cos_sim
x1	0.141421	0.2	0.1	0.999991
x2	0.670820	0.9	0.6	0.995752
x3	0.282843	0.4	0.2	0.999969
x4	0.223607	0.3	0.2	0.999028
x5	0.608276	0.7	0.6	0.965363

The table above shows the distance and similarities

```
In [81]: # shows the rank of similiarity for each of the measures
         df.rank()
```

```
Out[81]:     euclidean  manhattan  supremum  cos_sim
         x1        1.0        1.0       1.0      5.0
         x2        5.0        5.0       4.5      2.0
         x3        3.0        3.0       3.0      4.0
         x4        2.0        2.0       2.0      3.0
         x5        4.0        4.0       4.5      1.0
```

The table above shows the value ranks. Note that for measures of distance, lower values indicate higher similarity. For measures of similarity, higher values indicate higher similarity. Ranks by similiarity are:

euclidean: x1, x4, x3, x5, x2
manhattan: x1, x4, x3, x5, x2
supremum: x1, x4, x3, x5, x2
cosine similarity: x1, x3, x4, x2, x5

(b) Normalize the data set to make the norm of each data point equal to 1. Use Euclidean distance on the transformed data to rank the data points.

```
In [82]: # normalize a vector
         def normalize(obj):
             n = tuple(map(lambda x: x / LA.norm(obj),obj))
             return n
```

```
In [83]: # normalize data
         x_new_norm = normalize(x_new)
         x_all_norm = list(map(lambda x: normalize(x),x_all))
```

```
In [84]: # maps list into euclidean distance between each data point and the new data point
         eucl_norm = list(map(lambda x: dist(x,x_new_norm,2),x_all_norm))
```

```
In [85]: # puts mappings into a dataframe
         df_norm = pd.DataFrame({'euclidean_norm': eucl_norm},index=ind)
         df_norm
```

```
Out[85]:     euclidean_norm
         x1        0.004149
         x2        0.092171
         x3        0.007812
         x4        0.044085
         x5        0.263198
```

The table above shows the Euclidean distance between all of the data points and the new data point when all points are normalized to a length of 1.

```
In [86]: # shows rank
         df_norm.rank()
```

```
Out[86]:     euclidean_norm
         x1             1.0
         x2             4.0
         x3             2.0
         x4             3.0
         x5             5.0
```

The rank in order of similarity for normalized Euclidean distance:
normalized euclidean: x1, x3, x4, x2, x5

0.2 Chapter 3

[3.3] Exercise 2.2 gave the following data for the attribute age: 13, 15, 16, 16, 19, 20, 20, 21, 22, 22, 25, 25, 25, 25, 30, 33, 33, 35, 35, 35, 35, 36, 40, 45, 46, 52, 70.

(a) Use smoothing by bin means to smooth these data, using a bin depth of 3. Illustrate you steps. Comment on the effect of this technique for the given data.

To see the effect of binning on the data, it makes sense to first run an analysis on the data before binning.

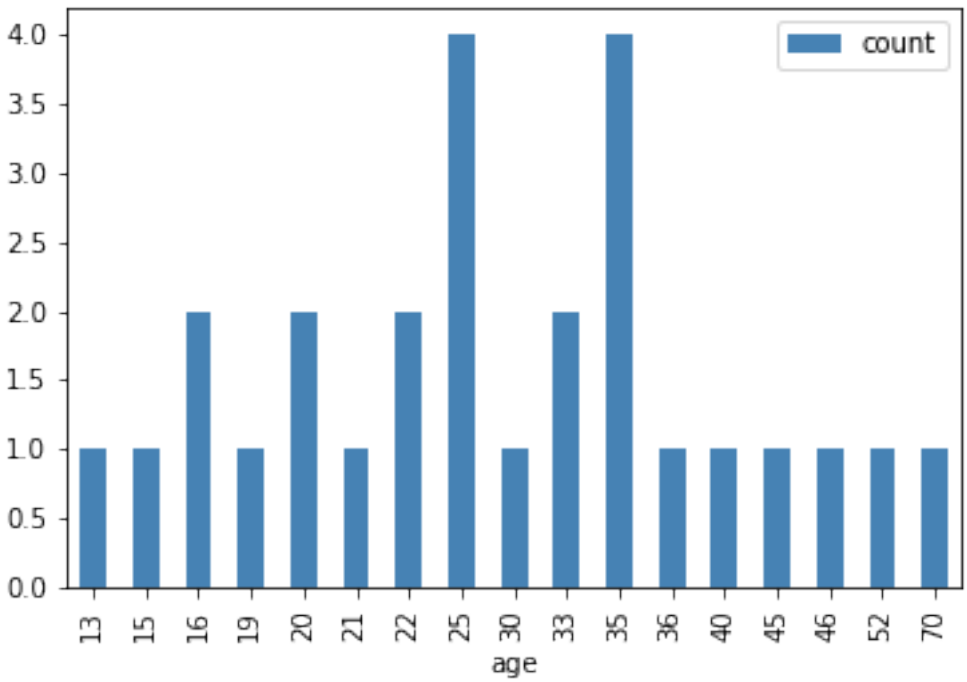
```
In [87]: import numpy as np
         ages = [13, 15, 16, 16, 19, 20, 20, 21, 22, 22, 25, 25, 25, 25,
                 30, 33, 33, 35, 35, 35, 35, 36, 40, 45, 46, 52, 70]
         mn = np.mean(ages)
         round(mn,2)

Out[87]: 29.96
```

The mean of this data set is 29.96

```
In [88]: md = np.median(ages)
         round(md, 2)

Out[88]: 25.0
```

N = 27, so the median value is 25.

```
In [89]: import pandas as pd
         mo = [[x,ages.count(x)] for x in set(ages)]
         df = pd.DataFrame(mo, columns=['age', 'count']).sort_values(by='age')
         df.plot.bar(x='age', y='count', color='steelblue')

Out[89]: <matplotlib.axes._subplots.AxesSubplot at 0x1209faa90>
```



The mode of this data set is 25 and 35. Based on the plot above, the plot appears to be bimodal in nature.

```
In [90]: midrange = (max(ages) - min(ages))/2
         midrange

Out[90]: 28.5
```

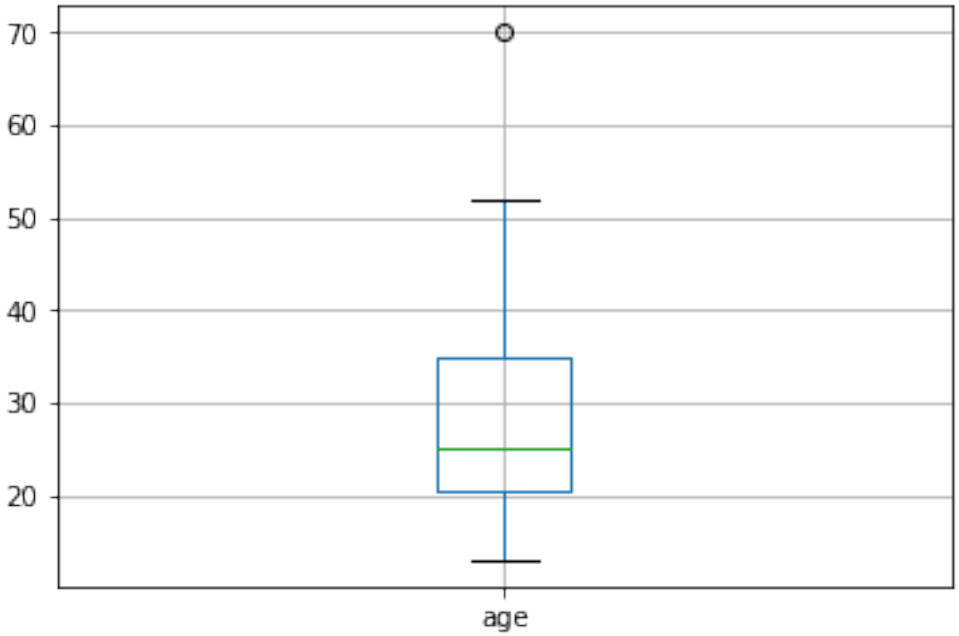
The midrange is defined as the average of the maximum and the minimum values of a data set. In this data set, the midrange is 28.5.

```
In [91]: df2 = pd.DataFrame(ages, columns=['age'])
         df2.describe()

Out[91]:              age
         count  27.000000
         mean   29.962963
         std    12.942124
         min    13.000000
         25%    20.500000
         50%    25.000000
         75%    35.000000
         max    70.000000
```

The five number summary is:
Min = 13.0

First quartile (Q1) = 20.5
Median (Q2) = 25.0
Third quartile (Q3) = 35.0
Max = 70

```
In [92]: df2.boxplot()

Out[92]: <matplotlib.axes._subplots.AxesSubplot at 0x120aa2400>
```



The boxplot is shown above.

Now we can bin the data and see the affect it has on the analysis. Details on how the bins were calculated is shown in the code comments.

```
In [93]: import math
         ages = [13, 15, 16, 16, 19, 20, 20, 21, 22, 22, 25, 25,
                 25, 25, 30, 33, 33, 35, 35, 35, 35, 36, 40, 45, 46, 52, 70]

         # bin the data
         def bin_data(data, width):
             # make a copy of the data
             data_copy = data

             #calculate the number of bins based on width param
             bins = math.ceil(len(data) / width)

             # loop through the bins
             for i in range(bins):
```

```
            # for each bin, set the sum to 0
            sum = 0

            # for each value within each bin,
            for j in range(width):

                # sum the values in the bin
                sum = sum + data[width * i + j]

            # get the bin average by dividing by width
            avg = sum / width

            # print the bins and values
            print('bin',i+1,': ',(str(round(avg,2))+', ')*3)

            # replace each value in data_copy with the average of the bin
            for j in range(width):
                data_copy[width * i + j] = avg

        # return the list of bin means
        return data_copy

    bin_ages = bin_data(ages,3)
```

```
bin 1 :  14.67, 14.67, 14.67,
bin 2 :  18.33, 18.33, 18.33,
bin 3 :  21.0, 21.0, 21.0,
bin 4 :  24.0, 24.0, 24.0,
bin 5 :  26.67, 26.67, 26.67,
bin 6 :  33.67, 33.67, 33.67,
bin 7 :  35.0, 35.0, 35.0,
bin 8 :  40.33, 40.33, 40.33,
bin 9 :  56.0, 56.0, 56.0,
```

```
In [94]: mn = np.mean(bin_ages)
         round(mn,2)
```

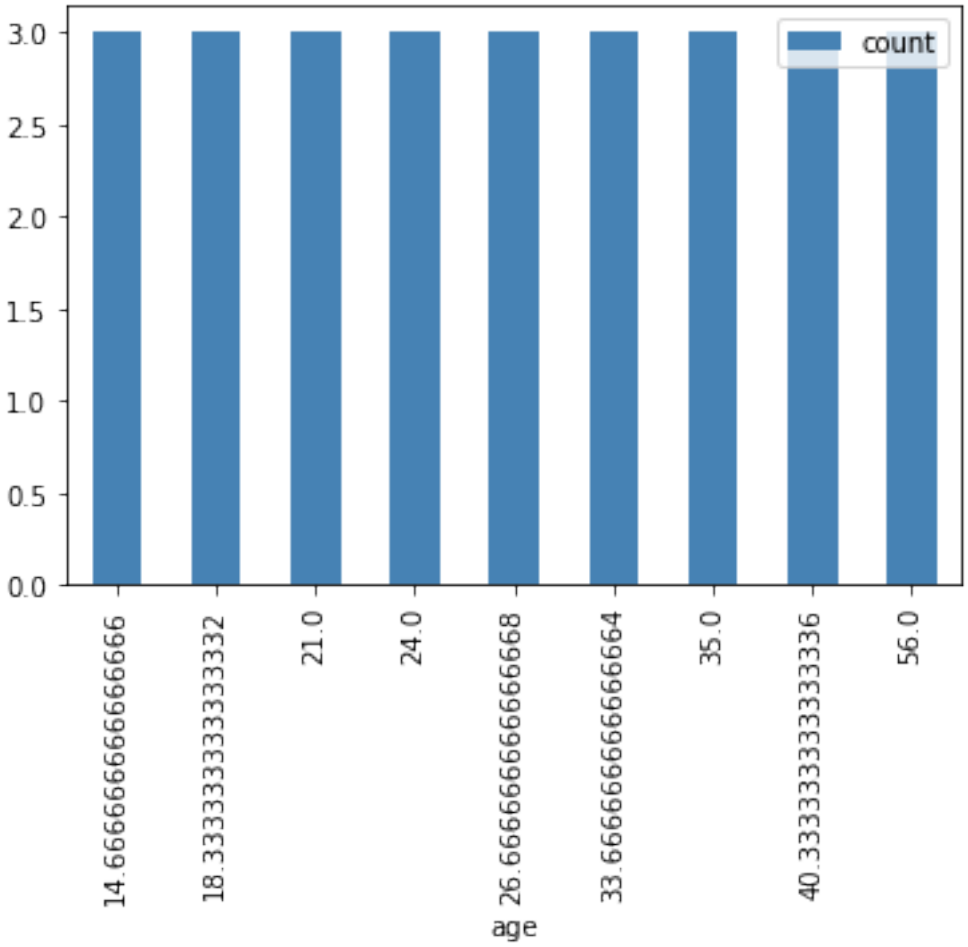
```
Out[94]: 29.96
```

```
In [95]: md = np.median(bin_ages)
         round(md, 2)
```

```
Out[95]: 26.67
```

```
In [96]: mo = [[x,bin_ages.count(x)] for x in set(bin_ages)]
         df3 = pd.DataFrame(mo, columns=['age', 'count']).sort_values(by='age')
         df3.plot.bar(x='age', y='count', color='steelblue')
```

Out[96]: <matplotlib.axes._subplots.AxesSubplot at 0x120b6d390>



In [97]:
```
midrange = (max(bin_ages) - min(bin_ages))/2
midrange
```

Out[97]: 20.666666666666668

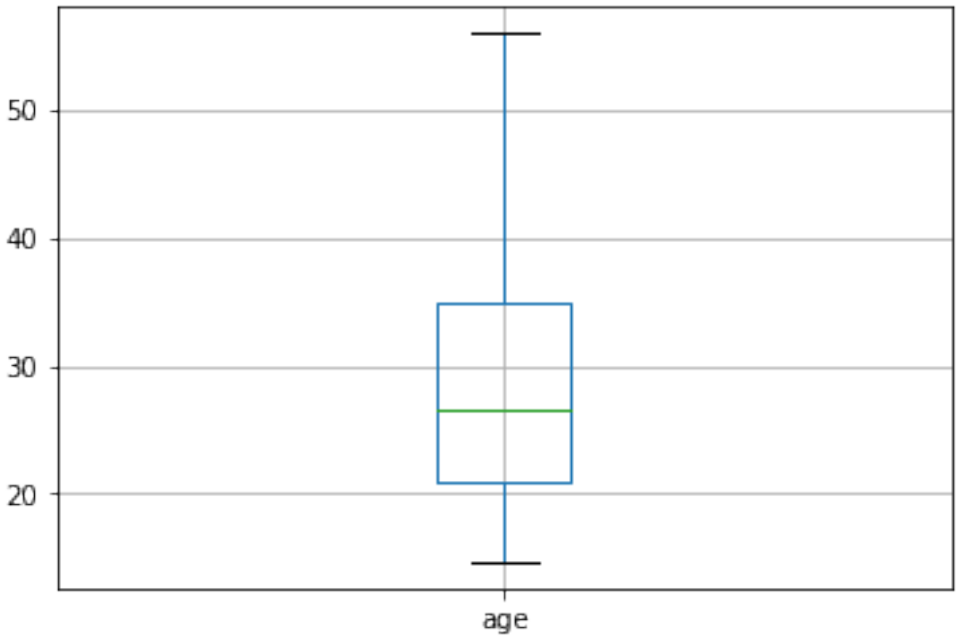
In [98]:
```
df4 = pd.DataFrame(bin_ages, columns=['age'])
df4.describe()
```

Out[98]:

	age
count	27.000000
mean	29.962963
std	12.343839
min	14.666667
25%	21.000000
50%	26.666667
75%	35.000000
max	56.000000

The five number summary comparison (binned vs. not binned) is:
Min = 14.67 vs. 13.0
First quartile (Q1) = 20.5 vs. 21.0
Median (Q2) = 25.0 vs. 26.7
Third quartile (Q3) = 35.0 vs. 35.0
Max = 70 vs. 56

```
In [99]: df4.boxplot()

Out[99]: <matplotlib.axes._subplots.AxesSubplot at 0x120aa2278>
```



The most material effect the binning has on the data is smoothing out the one outlier on the high end (70). The mean will not change, and in this case, the Q3 number doesn't change either. The Q1 and Q2 numbers change slightly. Since the data is smoothed, it is not clear now from the data that this is a bi-modal distribution.

(b) How might you determine outliers in the data?

For almost any data set, clustering can be used to identify outliers. Per the textbook, "Outliers may be detected by clustering, for example, where similar values are organized into groups or 'clusters'." Histograms and other data visualization techniques can be used to indentify outliers as well. When dealing with a univariate numeric data set, a boxplot can be used to visulaize outliers that fall outside of 1.5x the interquartile range. Points outside this range are generally considered outliers. If the population is assumed to be normally distributed, the standard deviation of the data can be calculated. Within this data set, points that fall outside:

$$\bar{x} \pm 3\sigma$$

are generally considered outliers.

(c) What other methods are there for data smoothing?

`Binning`: In addition to smoothing by bin means, there are other binning methods to smooth data. These include smoothing by median (where each bin value is replaced by the bin median), and smoothing by bin boundaries (the min and max are identified as bin boundaries and each value is then replaced by the closest value). Bins may be set to equal width, where the range of each bin is equal.

`Regression`: Data smoothing done by regression conforms data values to a function. Per the text: "Linear regression involves finding the best line to fit two attributes so that one attribute can be used to predict the other."

`Outlier analysis`: Outliers may be detected that fall outside of a data cluster and can be smoothed out by analyzing the clusters.

[3.5] What are the value ranges of the following normalization methods?

(a) Max-min normalization

The max-min normalization performs a linear transformation on the original data. The transformed range is typically [0,1] however it does not have to be. The range will be:

$$[newMin_A, newMax_A]$$

(b) z-score normalization

The range will depend on the range of the data, and can be calculated as:

$$[\frac{A_{min} - \bar{A}}{\sigma_A}, \frac{A_{max} - \bar{A}}{\sigma_A}]$$

(c) z-score normalization using the neam absolute deviation instead of standard deviation

This will be the same as above only normalizing by mean absolute deviation (s) instead of the standard deviation (sigma). The difference: the mean absolute deviation is more robust to outliers than the standard deviation because the deviations are not squared. The range will be:

$$[\frac{A_{min} - \bar{A}}{s_A}, \frac{A_{max} - \bar{A}}{s_A}]$$

(d) normalization by decimal scaling

Normalization by decimal scaling moves the decimal points of the values of an attribute of A. Since the data is scaled so such that the absolute max value is scaled to < 1. Therefore, the range will always be:

$$(-1, 1)$$

[3.6] Use these methods to normalize the following group of data: 200, 300, 400, 600, 1000

(a) min-max normalization by setting min=0, max=1

```
In [100]: data = (200, 300, 400, 600, 1000)

          def normalize(pt, mx, mn):
              return (pt - mn) / (mx - mn)

          def mm_norm(data, low, high):
              spread = (high - low)
              mx = max(data)
              mn = min(data)
```

```
            transform = list(map(lambda x: normalize(x,mx,mn)*spread,data))
            return transform

        mm_norm(data, 0, 1)
```

```
Out[100]: [0.0, 0.125, 0.25, 0.5, 1.0]
```

(b) z-score normalization

```
In [101]: def z_norm(data):
            mu = np.mean(data)
            sigma = np.std(data)
            transform = list(map(lambda x: (x-mu)/sigma,data))
            return transform

        z_norm(data)
```

```
Out[101]: [-1.0606601717798212,
           -0.7071067811865475,
           -0.35355339059327373,
           0.35355339059327373,
           1.7677669529663687]
```

(c) z-score normalization using mean absolute deviation

```
In [102]: from functools import reduce
        def mad(data):
            mad = reduce(lambda i,j: i+j, list(map(lambda x: abs(x - np.mean(data)),data))) /
            return mad

        def z_norm_mad(data):
            transform = list(map(lambda x: (x-np.mean(data))/mad(data),data))
            return transform

        z_norm_mad(data)
```

```
Out[102]: [-1.25,
           -0.8333333333333334,
           -0.4166666666666667,
           0.4166666666666667,
           2.0833333333333335]
```

(d) normalization by decimal scaling

```
In [103]: def norm_dec(data, j):
            transform = list(map(lambda x: (x/10**j),data))
            return transform

        # j = 4 because the formula requires abs(max)/10^j < 1, not <=1
        norm_dec(data, 4)
```

```
Out[103]: [0.02, 0.03, 0.04, 0.06, 0.1]
```

[3.7] Using the data for age given in exercise 3.3, answer the following:
(a) Use min-max normalization to transform the value 35 for age onthe the range [0.0,1.0]

```
In [104]: mm_norm(ages, 0, 1)[17]
```

```
Out[104]: 0.4596774193548387
```

(b) Use z-score normalization to tranform the value 35 for age onto the range [0.0,1.0]

```
In [105]: z_norm(ages)[17]
```

```
Out[105]: 0.3057603609074146
```

(c) Use normalization by decimal scaling to transform the value 35 for age.

```
In [106]: norm_dec(ages,2)[17]
```

```
Out[106]: 0.33666666666666667
```

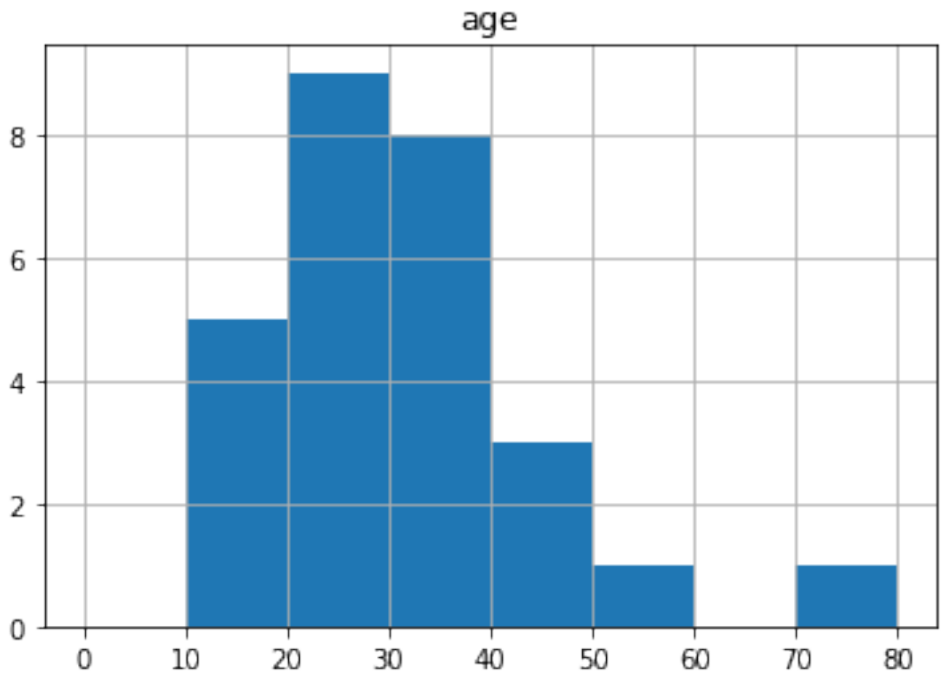
(d) Comment on the method you would prefer to use for the given data, giving reasons as to why.

With this data set, the most logical transformation seems to be decimal scaling. Min-max normalization would constrain all future data entries to between 13 and 70. Any data entered outside of this range would result in an out of bounds error. The process of normalizing the data using the z-score normalization process is computationally expensive and doesn't appear to add much information to the data. Typically this process is helpful when the population max/min are unknown, or when there are outliers that dominate the min-max normalization. This data set is a good candidate for decimal scaling because no information is lost in the transformation process but data is normalized to (0,1) with a very simple transformation as long as the max age is <100.

[3.11] Using the data for age given in exercise 3.3
(a) Plot an equal width histogram of width 10

```
In [107]: df2.hist(bins=[0, 10, 20, 30, 40, 50, 60, 70, 80])
```

```
Out[107]: array([[<matplotlib.axes._subplots.AxesSubplot object at 0x1207c8828>]],
                dtype=object)
```



(b) Sketch examples of each of the following sampling techniques: SRSWOR, SRSWR, cluster sampling, stratified sampling. Use samples of size 5 and the strata "youth", "middle-aged", "senior"

```
In [108]: import random

          def srswor(data,s):
              sample = []
              used = []
              for i in range(s):
                  rand = random.randint(0, len(data)-1)
                  while (rand in used):
                      rand = random.randint(0, len(data)-1)
                  sample.append(data[rand])
                  used.append(rand)
              return sample

          srswor(ages,5)

Out[108]: [33.666666666666664, 14.666666666666666, 21.0, 56.0, 14.666666666666666]

In [109]: def srswr(data,s):
              sample = []
              for i in range(s):
                  rand = random.randint(0, len(data)-1)
                  sample.append(data[rand])
```

```
            used.append(rand)
        return sample

    srswor(ages,5)
```

```
Out[109]: [14.666666666666666, 24.0, 33.666666666666664, 21.0, 14.666666666666666]
```

Cluster sample:

Stratified sample:

| before sampling | | after sampling | |--------|--------| |--------|--------| | 13 | youth | | 13 | youth | | 15 | youth | | 20 | youth | | 20 | youth | | 32 | middle_aged | | 27 | youth | | 39 | middle_aged | | 30 | middle_aged | | 40 | middle_aged | | 32 | middle_aged | | 42 | middle_aged | | 39 | middle_aged | | 56 | middle_aged | | 39 | middle_aged | | 70 | senior | | 40 | middle_aged | | 42 | middle_aged | | 42 | middle_aged | | 50 | middle_aged | | 56 | middle_aged | | 60 | middle_aged | | 70 | senior | | 80 | senior |

Propose an algorithm for the following:

(a) The automatic generation of a concept of hierarchy for nominal data based on the number of distict values of attributes in the given schema.

```
In [110]: from heapq import heappush, heappop

    # read in data from file and import sample columns
    # this sample file reads in data for every school K-12 in the US
    df = pd.read_csv('schools.csv',usecols=['STATENAME',
                                            'SCH_NAME',
                                            'LEA_NAME',
                                            'MSTREET1',
                                            'MCITY',
                                            'LEVEL'])

    # put the column names in a list
    column_names = list(df.columns.values)

    # create a 2-D array of values
    data = df.values

    # get the number of columns in the data set
    cols = np.shape(data)[1]

    # create an empty heap to store the unique count values
    h = []

    for i in range(cols):
        # slice the data by columns and get the length of the unique values in each colu
        length = len(set(data.T[i]))
        header = column_names[i]
        # push the unique values and column header to the heap
        heappush(h, (length, header))
```

```
        #print the heap (from low to high)
        for i in range(cols):
            print(heappop(h))
```

```
(8, 'LEVEL')
(56, 'STATENAME')
(14453, 'MCITY')
(17305, 'LEA_NAME')
(85247, 'MSTREET1')
(89637, 'SCH_NAME')
```

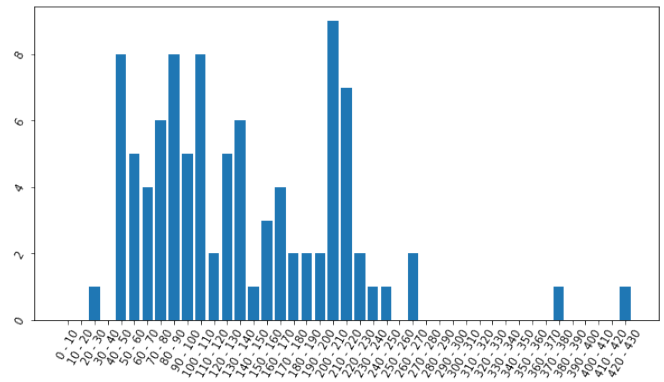
The data set I used is the data for every K-12 public school in the US. The steps were loading the data, transforming the data to an array, slicing it by columns, finding the number of unique values in each column (using set in Python which uses hashing), storing those in a heap (for efficiency) and printing in order. The final step is to add a layer of judgement. In this case, 'LEVEL' seems to be the broadest category. However it doesn't really fit into the 'location' schema because it describes the 'type' of school (primary, elementary, middle, high school). However, STATE CITY DISTRICT ADDRESS SCHOOL as a heirarchy makes sense. The ADDRESS SCHOOL is noteworthy because it suggests that a number of schools may be co-located in one building or campus.

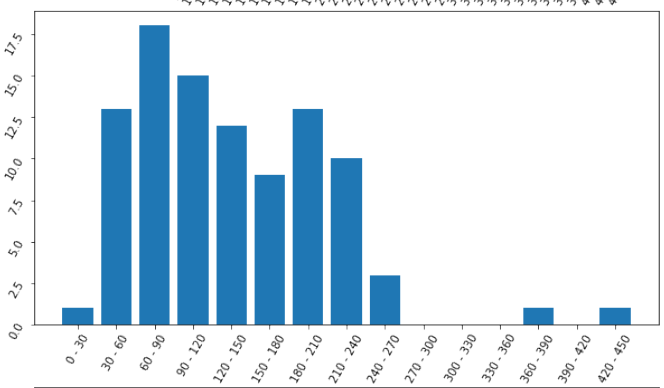
(b) The automatic generation of a concept hierarchy for numeric data based on the equal-width partitioning rule.

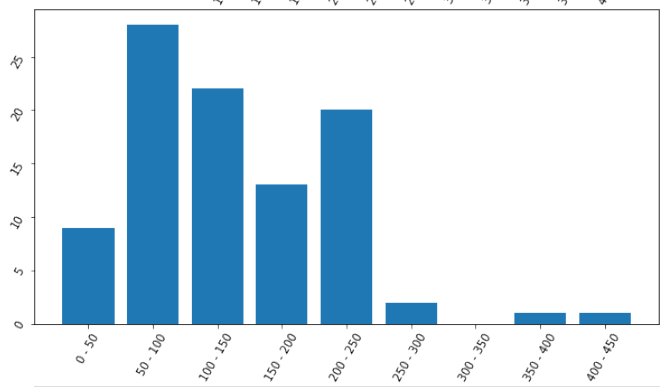
```
In [111]: # import a data set of the average ticket price of each of the 96 NFL game in 2014
          df = pd.read_csv('prices.csv')

          # get the list of all prices
          prices = df['AvgTP'].tolist()
```

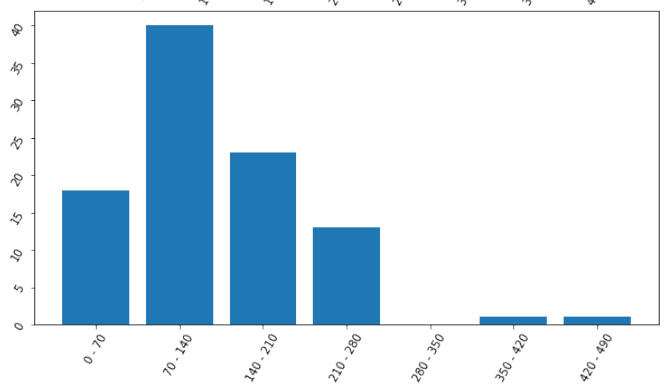
```
In [112]: import matplotlib.pyplot as plt
          import numpy as np

          # calculate equal width frequency
          # params: data, starting point, width of bin
          def eqw_bin(data, start, width):
              # sort the data
              data.sort()
              # create a new array for the bin names
              x = []
              # create a new array for the bin frequency
              y = []
              # start where user specifies
              curr_min = start
              # start the bin count
              count = 0
              # for each value in the data set
              for i in range(len(data)):
```

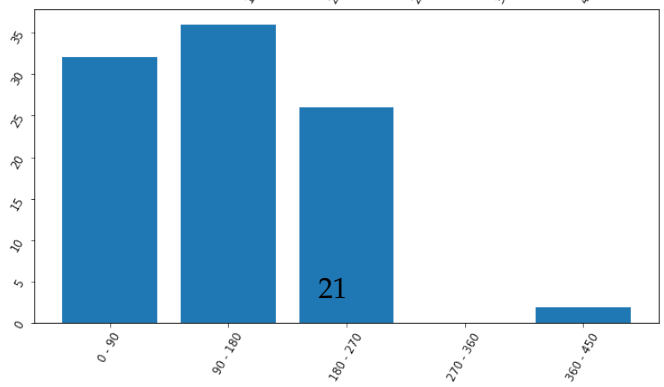
```
            # if the value is in the current bin, count it
            if (data[i] < curr_min + width):
                count += 1
            # if not, append the current count and name to appropriate array
            # increment the bin min value to the next bin (current + width)
            # reset the counter for this bin
            # wrap in a while loop to accomodate empty bins
            # once the correct bin is found, incremenbt the counter
            else:
                while (data[i] >= curr_min + width):
                    y.append(count)
                    x.append(str(int(curr_min))+' - '+str(int(curr_min+width)))
                    curr_min += width
                    count = 0;
                count += 1
            # handle the edge case of the last value in the array
            if(i == len(data)-1):
                y.append(count)
                x.append(str(int(curr_min))+' - '+str(int(curr_min+width)))
        # return the two arrays
        return (x,y)

    # run the above equation multiple times based on a specified min_ and max_width and
    def eqw_rec(data, start, min_width, max_width, levels):
        step = (max_width - min_width) / levels
        fig, axs = plt.subplots(levels, 1, figsize=(10, 30))
        for i in range(levels):
            new_data = eqw_bin(data, start, min_width+i*step)
            axs[i].bar(new_data[0],new_data[1])
            axs[i].tick_params(rotation=60)


    eqw_rec(prices, 0, 10, 110, 5)
```


0 - 10
10 - 20
20 - 30
30 - 40
40 - 50
50 - 60
60 - 70
70 - 80
80 - 90
90 - 100
100 - 110
110 - 120
120 - 130
130 - 140
140 - 150
150 - 160
160 - 170
170 - 180
180 - 190
190 - 200
200 - 210
210 - 220
220 - 230
230 - 240
240 - 250
250 - 260
260 - 270
270 - 280
280 - 290
290 - 300
300 - 310
310 - 320
320 - 330
330 - 340
340 - 350
350 - 360
360 - 370
370 - 380
380 - 390
390 - 400
400 - 410
410 - 420
420 - 430


0 - 30
30 - 60
60 - 90
90 - 120
120 - 150
150 - 180
180 - 210
210 - 240
240 - 270
270 - 300
300 - 330
330 - 360
360 - 390
390 - 420
420 - 450


0 - 50
50 - 100
100 - 150
150 - 200
200 - 250
250 - 300
300 - 350
350 - 400
400 - 450


0 - 70
70 - 140
140 - 210
210 - 280
280 - 350
350 - 420
420 - 490


0 - 90
90 - 180
180 - 270
270 - 360
360 - 450