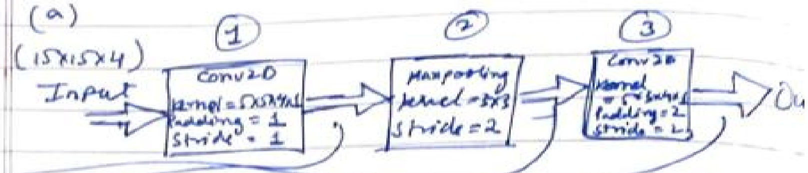
SECTION- A

Section-A

Q1. (a)



(i) For conv. layer 1, $w = \left(\frac{15-5+(2\times1)}{1}\right)+1 =$

" ~~for maxpool~~, " , $h = \left(\frac{15-5+2\times1}{1}\right)+1 =$

13X13

for maxpool layer, $w_1 = \left(\frac{13-3}{2}\right)+1 = 6$

" , $h_1 = \frac{(13-3)}{2}+1 = 6$

6x6

After last conv. layer,

$$w_2 = \left(\frac{6-5+(2\times2)}{2}\right)+1 = [3.5] \simeq 3$$

$$h_2 = \left(\frac{6-3+(2\times2)}{2}\right)+1 = (4.5) \simeq 4$$

$\therefore$ output layer has dimensions $= [3\times4]$

DATE: / /
PAGE:

(ii) Significance of pooling in CNN:-

- Reduces the dimensionality of the feature maps.
- helps to achieve translation invariance
- helps in parameter sharing
- Reduces the computation cost.

(iii) For conv layer 1, => $n_1 = (5 \times 5 \times 4 \times 1) \times 1$
$\boxed{= 100}$

" " 2, => $n_2 = (5 \times 3 \times 4 \times 1) \times 1$
$\boxed{= 60}$

For pooling layer = 0 learnable

∴ Total parameters (w/o bias) = 160

(b) (3, 12), (3, 7), (9, 6), (6, 10), (8, 7), (7, 6), (2, 13)

$D(a, b) = |x_2 - x_1| + |y_2 - y_1|$

Intial clusters $c_1 = (3, 12)$, $c_2 = (8, 7)$, $c_3 = (2, 13)$

Distance from

Point	c_1	c_2	c_3	Cluster $(\min(c_1, c_2, c_3))$
3,12	0	10	2	c_1
3, 7	5	5	7	c_1
9, 6	12	2	14	c_2
6,10	5	5	7	c_2
8, 7	10	0	12	c_2
7, 6	10	2	12	c_2
2,13	2	12	0	c_3

New centroids for, $c_1 = \left(\frac{3+3}{2}, \frac{12+7}{2}\right)$

$\boxed{c_1 = (3, 9.5)}$

for $c_2 = \left(\frac{9+6+8+7}{4}, \frac{6+10+7+6}{4}\right) = \boxed{(7.5, 7.25)}$

PAGE

for c_3 = (2, 13) → same

$c_1 = (3, 9.5)$

$c_2 = (7.5, 7.25)$

2nd Iteration $c_3 = (2, 13)$ (min(c_1,c_2,c_3))

Points	Dist from new c_1	Dist from new c_2	Dist from new c_3	Cluster
(3,12)	2.5	9.25	2	c_3
(3,7)	2.5	4.75	7	c_1
(9,6)	9.5	2.75	14	c_2
(6,12)	3.5	4.25	7	c_1
(8,7)	7.5	0.75	12	c_2
(7,6)	7.5	1.75	12	c_2
(2,13)	4.5	11.25	0	c_3

∴ New centres $c_1 = \left(\frac{3+6}{2}, \frac{7+10}{2}\right) = (4.5, 8.5)$

$c_2 = \left(\frac{9+8+7}{3}, \frac{6+7+6}{3}\right) = (8, 6.33)$

$c_3 = \left(\frac{3+2}{2}, \frac{13+12}{2}\right) = (2.5, 12.5)$

Ans

SECTION - C

A. Reading all the given datasets

```
In [2]: dd=pd.read_csv('/Users/vaibhavwali/Desktop/a4/Dataset Description.csv
        mt5k=pd.read_csv('/Users/vaibhavwali/Desktop/a4/more_than_50k.csv')
        ppln=pd.read_csv('/Users/vaibhavwali/Desktop/a4/population.csv')
        dd
        ppln.info()
        # ppln.head()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 199523 entries, 0 to 199522
Data columns (total 40 columns):
 #   Column    Non-Null Count   Dtype
---  ------    --------------   -----
 0   AAGE      199523 non-null  int64
 1   ACLSWKR   199523 non-null  object
 2   ADTIND    199523 non-null  int64
 3   ADTOCC    199523 non-null  int64
 4   AHGA      199523 non-null  object
 5   AHRSPAY   199523 non-null  int64
```

B and C.

```
#preprocessing of data
print(ppln.isnull().sum())
for i in ppln:
    check=ppln[i]
    check[check==' ?']=np.nan
```

```
print(ppln.isnull().sum())
print(ppln.shape)
ptgs=[]
print("----------------------")
for i in ppln:
    count=ppln[i].isnull().sum()
    ptgs.append((i,count/len(ppln)))

    if(ptgs[-1][1]>0.3):
        ppln=ppln.drop(i,axis=1)

print("Columns with more than 30% of missing data: ")
for i in ptgs:
    if(i[1]>0.3):
        print(i[0])
print ("----------------------")
print("Remaining columns: ")
print(ppln.isnull().sum())
print(ppln.shape)
```

In this section, we account for the missing data. We check if there are any "?" values, if yes we replace them with null values. Further, we check the number of missing values in each column and if those columns are missing more than 30% of the data then we drop those columns. This results in the dropping of 4 columns in both datasets.

```
SEOTR           0
VETQVA          0
VETYN           0
WKSWORK         0
YEAR            0
dtype: int64
(199523, 40)
----------------------
Columns with more than 30% of missing data:
MIGMTR1
MIGMTR3
MIGMTR4
MIGSUN
----------------------
Remaining columns:
AAGE           0
ACLSWKR        0
ADTIND         0
```

With the concept of bins, we bucketize the numerical data into categorical data. We manually check the columns which have numerical data. The columns are given in the below image:-

```
In [6]: print("Numerical data exists in these columns")
        for i in (ppln):
            if(type(ppln[i][0])!=type('a')):
                print(i)

Numerical data exists in these columns
AAGE
ADTIND
ADTOCC
AHRSPAY
CAPGAIN
CAPLOSS
DIVVAL
NOEMP
SEOTR
VETYN
WKSWORK
YEAR
```

These columns have numerical values and we bucketize them using pandas.cut, given below is a demonstration of how I have done it.

```
In [8]: # print(ppln['AAGE'])
        bins = [0,3,5,13,19,60,100]
        labels = ['AAGE infant','AAGE toddler','AAGE child','AAGE teenager','AAGE adult','AAGE senior citizen
        ppln['Age Category'] = pd.cut(ppln['AAGE'],bins,labels = labels)
        print(ppln['Age Category'].value_counts())
        ppln['Age Category'].value_counts().plot(kind = 'pie')
        ppln = ppln.drop(['AAGE'],axis=1)
        ppln.head()

        AAGE adult              108231
        AAGE senior citizen      30397
        AAGE child               25212
        AAGE teenager            16541
        AAGE infant               9653
        AAGE toddler              6650
        Name: Age Category, dtype: int64

Out[8]:
```

I have done this for every numerical column in the same way and further dropped the numerical column.

For Imputation, we replace the null data values (if there exists any) with the mode of that column. This is done in the following way:-

```
In [7]: for column in ppln.columns:
            ppln[column].fillna(ppln[column].mode()[0], inplace=True)
        print(ppln.isnull().sum())

        AAGE            0
```

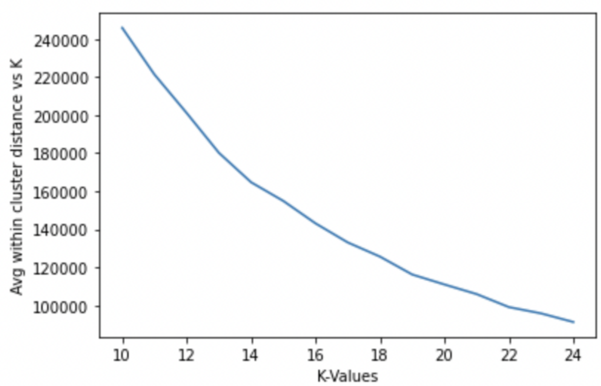
Lastly, for OneHotEncoding, we use the sklearn library to achieve that. In this, we change all the categorical data into numerical data for further calculation of the K-median (as it does not work well on categorical data). We achieve this in the following way:-

```
: #ONE HOT ENCODING
  column=[]
  for i in ppln:
      column.append(i)
  print(len(column))
  print(column)
  ohe = OneHotEncoder()
  feat_array = ohe.fit_transform(ppln[column]).toarray()
  print(feat_array)
  feat_labels = ohe.categories_
  feat_labels = np.hstack(feat_labels)
  # feat_labels = np.array(feat_labels).ravel()
  # print(feat_labels)
  new_feat1 = pd.DataFrame(feat_array,columns = feat_labels)
  new_ppln = pd.concat([ppln,new_feat1],axis = 1)
  new_ppln.head(10)

  36
  ['ACLSWKR', 'AHGA', 'AHSCOL', 'AMARITL', 'AMJIND', 'AMJOCC', 'ARACE', 'AREORGN', 'ASEX', 'AUNMEM', 'AUNTYPE', 'AWKST
  T', 'FILESTAT', 'GRINREG', 'GRINST', 'HHDFMX', 'HHDREL', 'MIGSAME', 'PARENT', 'PEFNTVTY', 'PEMNTVTY', 'PENATVTY', 'P
  CITSHP', 'VETQVA', 'Age Category', 'AHRSPAY Category', 'CAPGAINS Category', 'CAPLOSS Category', 'DIVVAL Category', '
  eeks Worked Category', 'ADTIND Category', 'ADTOCC Category', 'NOEMP Category', 'SEOTR Category', 'VETYN Category', '
  EAR Category']
  [[0. 0. 0. ... 0. 1. 0.]
   [0. 0. 0. ... 0. 0. 1.]
   [0. 0. 0. ... 0. 1. 0.]
   ...
   [0. 0. 0. ... 0. 1. 0.]
   [0. 0. 0. ... 0. 1. 0.]
   [0. 0. 0. ... 0. 0. 1.]]
```

D.

I have achieved K-medians clustering by implementing it in the utils.pynb file and further importing that file into our main file. We were advised to use the distance formula of:-
D = |x1-x2| + |y1-y2|. Hence, while implementing the class I have used the same formula. My graph of avg-within cluster distance vs no. of cluster graph is given below:-

```
In [24]: plt.xlabel("K-Values")
         plt.ylabel("Avg within cluster distance vs K")
         plt.plot(kv,lv)

Out[24]: [<matplotlib.lines.Line2D at 0x7fc5412a7070>]
```



As we can't see any prominent 'elbow' in the graph, I took my K value to be 21 for further analysis of data. I also did some pre-processing before calling my K-medians class. I did PCA and reduced the dimensions of my dataset to 50 since the original dataset (after one-hot encoding) had 414 columns, hence it significantly increased the computational time.

```python
class KMedians():

    def __init__(self,n_clusters=3,n_iters= 8,n_dimensions = 50):
        self.n_clusters=n_clusters
        self.n_iters=n_iters
        self.n_dimensions=n_dimensions
        self.centers=np.zeros((self.n_clusters,self.n_dimensions))

    def fit(self,data,plot=False):
        self.data=data
        self.centers=np.zeros((self.n_clusters,self.n_dimensions))
        self.randoms=random.sample(range(len(data)),self.n_clusters)

        for i in range(len(self.randoms)):
            self.centers[i]=data[self.randoms[i]]

        self.loss_store=[]
        self.iterations=[]

        for iterr in range(self.n_iters):
            self.distance={}
            self.iterations.append(iterr)
            for k in range(self.n_clusters):
                self.distance[k]=self.data-self.centers[k]
                self.distance[k]=np.absolute(self.distance[k])
                self.distance[k]=np.sum(self.distance[k],axis=1)
            self.min_distance=self.distance[0]

            for k in range(self.n_clusters):
                self.min_distance=np.minimum(self.min_distance,self.distance[k])

            self.loss_store.append(np.sum(self.min_distance))
            self.clusters={}

            for k in range(self.n_clusters):
                self.clusters[k]=(self.distance[k]==self.min_distance)
                self.clusters[k]=self.data[self.clusters[k]]

            for k in range(self.n_clusters):
```

Given above is the implementation of my KMedian functions. K-medians is an algorithm for cluster analysis clustering. It is a variant of k-means clustering where the median is calculated rather than the mean for each cluster.

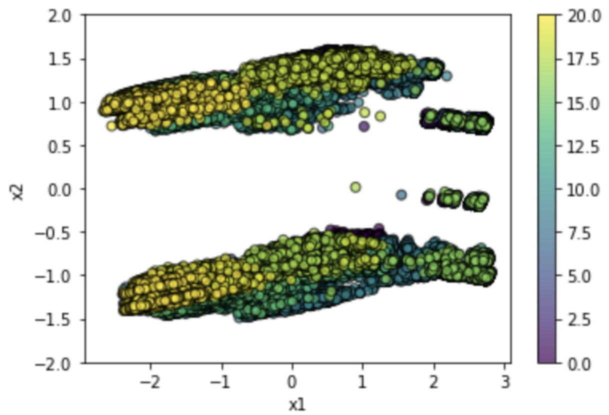
E and F.

I have done the exact same thing for the **more than 50k population** dataset as well. Just copied the code and changed whatever was required.

One of the initial differences that I noticed was that the dataset obtained after one-hot encoding had much lesser columns (387). This means that we have fewer data in the morethan50k file as compared to the general population file.

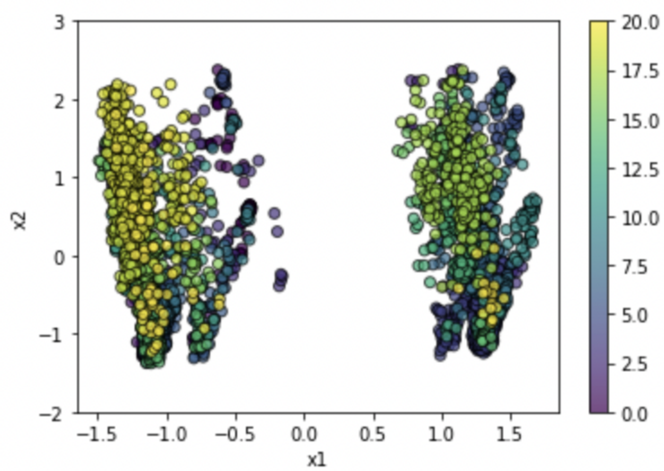
Given below is the scatter plot of the general population dataset

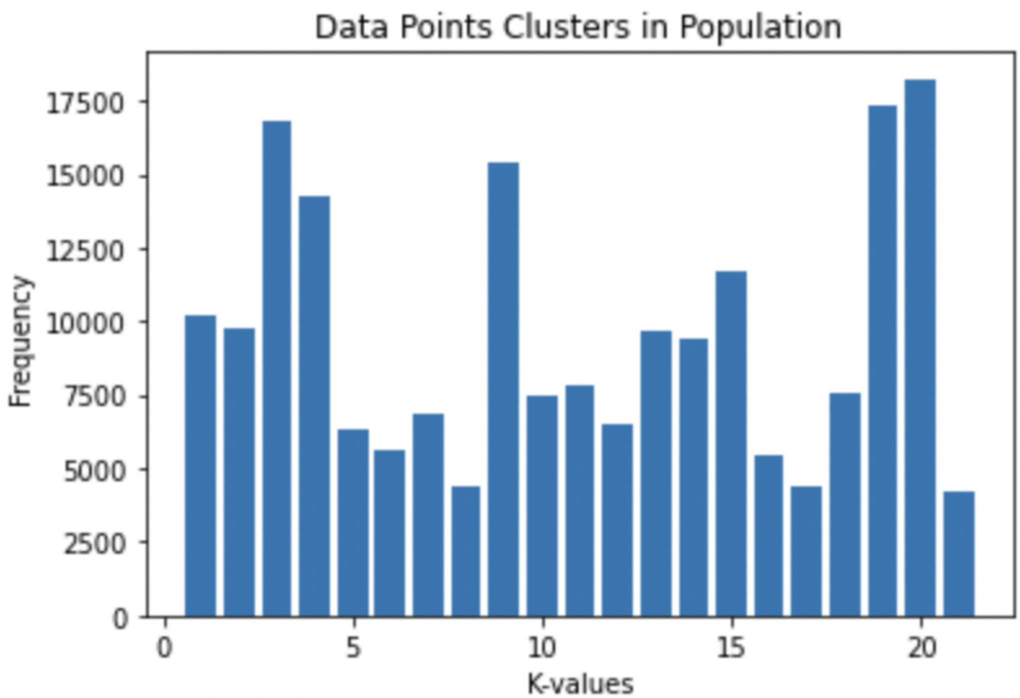
[ ]

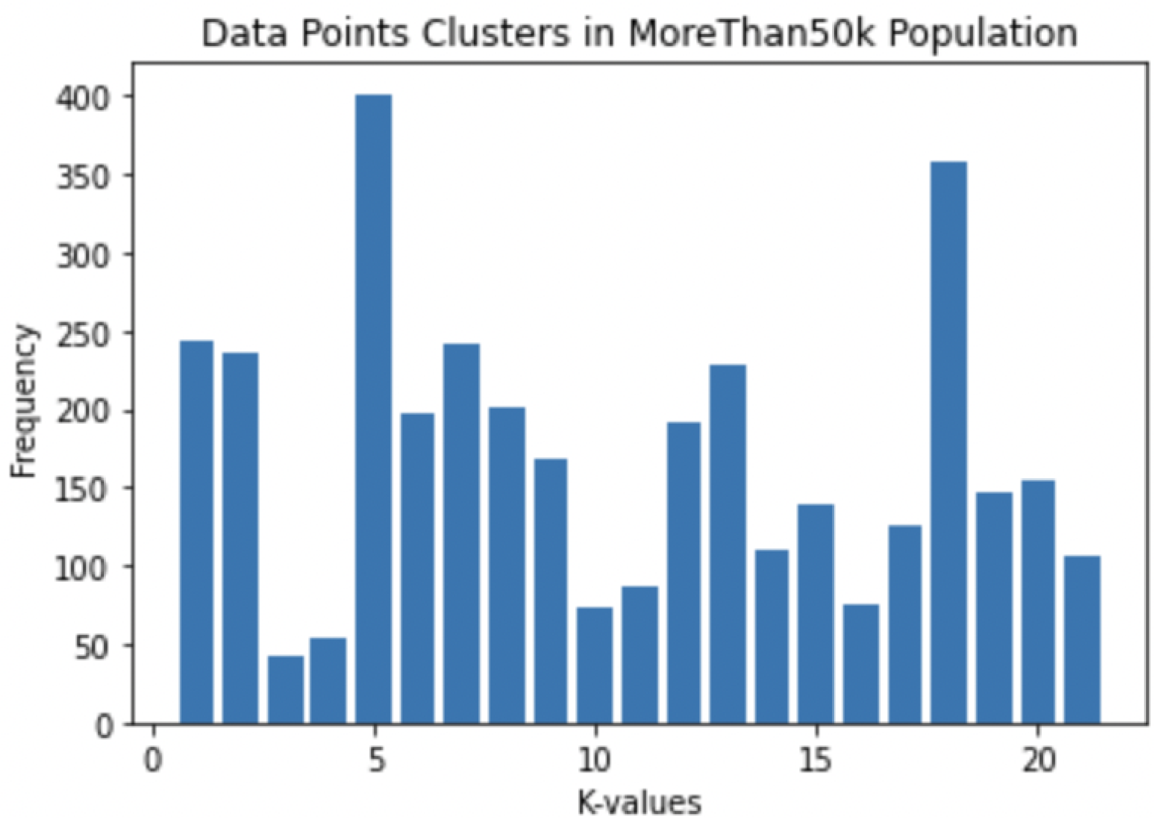


Given below is the scatterplot of more than 50k dataset

[ ]




Data Points Clusters in Population
Frequency
K-values
0
2500
5000
7500
10000
12500
15000
17500
0
5
10
15
20


Data Points Clusters in MoreThan50k Population
Frequency
K-values
0
50
100
150
200
250
300
350
400
0
5
10
15
20

We can clearly see the difference between them in each cluster.

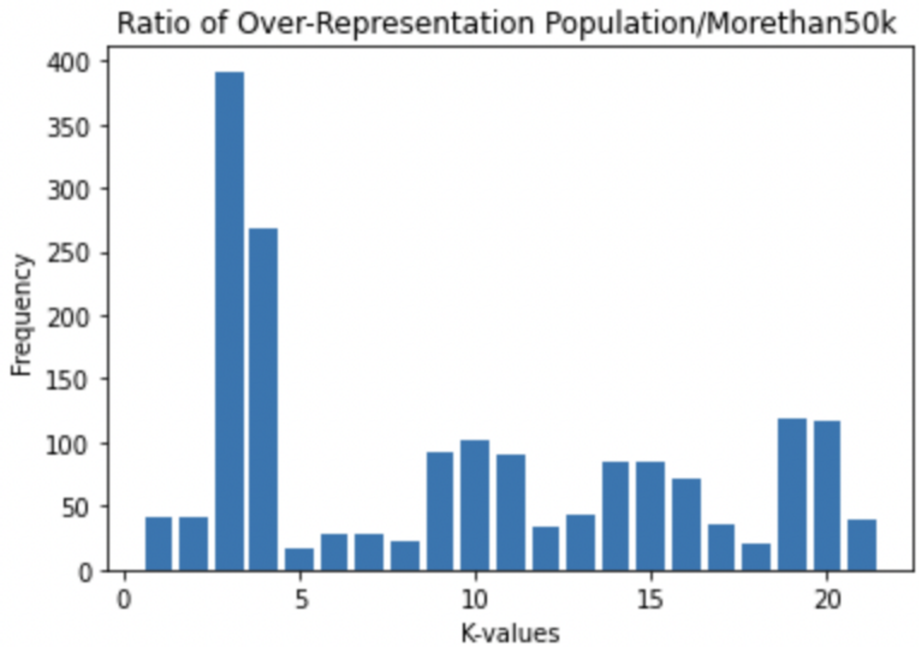
From the above plots, we can see the density of data in each cluster. The colorbar represents the number of clusters.

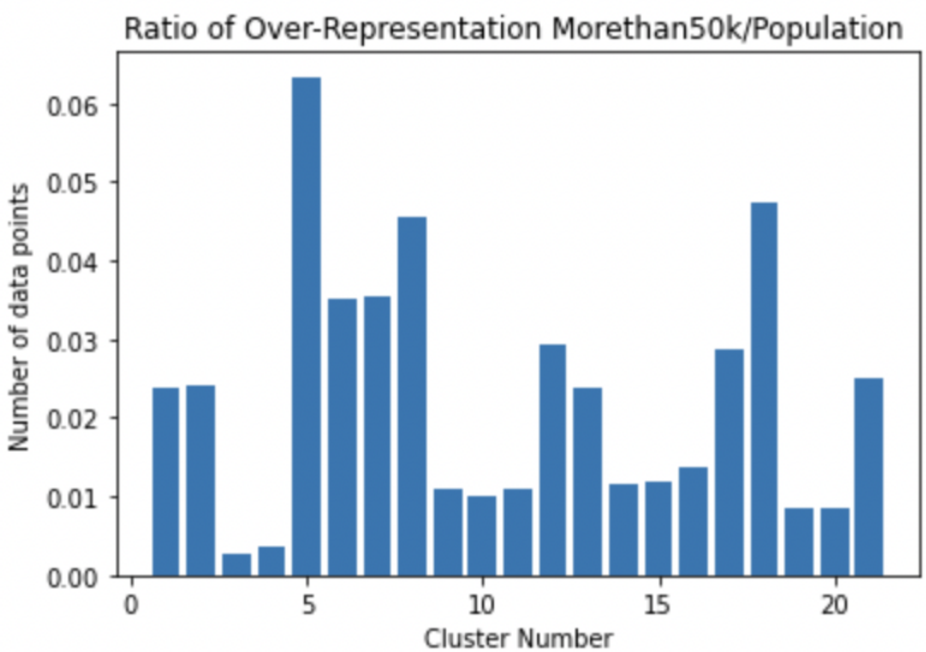
Some clusters are highly concentrated at the centre some are not.

In the bar plot for the general population dataset, the highest number of points are present in clusters:- 4,19 and 20.

In the bar plot for the more than 50k population dataset, the highest number of points are present in clusters:- 5 and 18.

To calculate the clusters' over-represnation in general vs the more dataset and vice versa I have made more bar plots as given below:-





The bar plots above give us our desired output, i.e. highest over-represenation is in 3 and 4 for the general population vs morethan50k whereas in the case of morethan50k vs the general population the highest over-represenation is in 5 and 18.

We sorted the dataset after inverse transformation and then took the top values in the first principal component to analyze the values for the centroid.

We can compare the two bar plots for each cluster respectively and see the difference in each cluster so as to check the kind of people that are over-represented in the morethan50k population vs the general population dataset

SECTION - B

The Convolution Layer

```
class Convulation:

    def __init__(self, nooffilt):

        self.nooffilt = nooffilt
        self.filt = np.random.randn(nooffilt, 3, 3)/9

    def zero(self, image):
        image=np.pad(image, (1, 1), 'constant', constant_values=(0, 0))


    def window(self,image):
        h,w = image.shape
        for i in range(1,h-4):
            for j in range(1,w-4):
                imgwdw = image[i:i+3, j:j+3]
                yield imgwdw, i, j

    def forward(self, input):
        self.last_input = input
        h,w = input.shape
        output = np.zeros((h-2, w-2, self.nooffilt))
        self.zero(input)
        for imgwdws, i, j in self.window(input):
            output[i, j] = np.sum(imgwdws * self.filt)
        return output

    def backprop(self, b1_out):

        b1_filt = np.zeros(self.filt.shape)
        self.zero(self.last_input)
        for imgwdw, i, j in self.window(self.last_input):
            for f in range(self.nooffilt):
                b1_filt[f] += b1_out[i,j,f] * imgwdw

        self.filt = self.filt - (learn_rate * b1_filt)
```

Given above is the scratch implementation of my Convolution Layer having all the desired functions:-

The init function:- Constructor initialising all the required features

The zero function:- It adds a layer of zero padding to all the 4 sides of the array, i.e. increasing the size of the image input by (2,2)

The window function:- This function simply iterates over small chunks of the input image, in my case I have taken chunks of 3 X 3, and stores those chunks in another array.

The forward function:- This function helps us to get the output from the convolution layer, it initialises an array of zeros first as an output array and adds a filter (kernel) to the input array to give us the output array. It operates according to the formula:- dim(Output) = [dim(Input) - dim(Kernal) + dim(padding)]/dim(Stride) + 1.

The backprop function:- This is basically used to propagate the Convolution layer in a backwards manner.

The Pooling Layer

```python
class Pooling:

    def amask(x):
         mask = x == np.max(x)

         return mask


    def distribute(self, image):
        h, w, _ = image.shape

        hnew = h // 2
        wnew = w // 2

        for i in range(hnew):
            for j in range(wnew):
                imgwdw = image[i*2:i*2+2, j*2:j*2+2]
                yield imgwdw, i, j


    def forward(self, input):
        self.last_input = input
        h, w, nooffilt = input.shape
        output = np.zeros((h//2, w//2, nooffilt))
        for imgwdw, i, j in self.distribute(input):
            output[i,j] = np.amax(imgwdw)
        return output

    def backprop(self, back_out):
        back_input = np.zeros(self.last_input.shape)
        for imgwdw, i, j in self.distribute(self.last_input):
            h, w, f = imgwdw.shape
            amax = np.amax(imgwdw, axis=(0,1))
            k = 0
            while k < h:
                l =0
                while l < w:
                    m = 0
                    while m < f:
                        if(imgwdw[k,l,m] == amax[m]):
                            back_input[i*2+k, j*2+l ,m] = back_out[i, j, m]
                            break;
```

Given above is the scratch implementation of my Pooling Layer having all the desired functions:-

The amask function:- It is used to keep track of the maximum of the matrix as I have taken max pooling into account.

The distribute function:- This function basically distributes values to all the required parameters appropriately after doing some amount of processing.

The forward function:- This operates in a similar fashion as the forward function in the conv layer.

The backprop function:- Operates in a similar way as the conv backprop function (takes the gradient descent into account to calculate the loss).

The Main Function

```
train_images = mnist.train_images()[:500]
train_labels = mnist.train_labels()[:500]
Convulation = Convulation(4)
pool = Pooling()

def distribute_values(input,input_len,nodes):
        weights = np.random.randn(input_len, nodes)
        biases = np.ones(nodes)
        input = input.flatten()
        totals = np.dot(input, weights) + biases
        return totals

def analyse(input,input_len,nodes):
        totals = distribute_values(input,input_len,nodes)
        lin = np.dot(totals,1)
        out = np.sum(lin, axis=0)
        return out

def verify(image, label):

    out = Convulation.forward((image/255))
    out = pool.forward(out)
    out = analyse(out,13 * 13 * 4, 10)


    if(np.argmax(out) == label):
        return 1
        s=s+1

    else:

      return 0
      l=l+1
s = 0
t = 0
for i in range(len(train_images[:1000])):

    loss = verify(train_images[i],train_labels[i])
    if(loss == 0):
```

The distribute values function basically does some preprocessing on the data(like the addition of linear activation function in the forward pass).

The analyse and verify functions help in doing some amount of calculations and training on our dataset. We have considered the mnist dataset. The final accuracy in my case came out to be around 90%.

```
Incorrect Classification:
Correct Classification:
Correct Classification:
Correct Classification:
Correct Classification:
Correct Classification:
Correct Classification:
Correct Classification:
Correct Classification:
Correct Classification:
Correct Classification:
450
50
Accuracy:  90.0 %
```