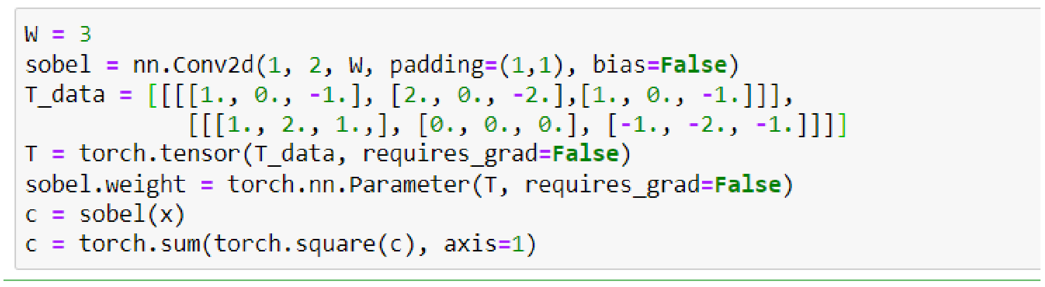
Home Assignment 6
Machine Learning A

FEDERICO FIORIO

October 13, 2022

1 Convolutional Neural Networks (50 points)

1.1 Sobel filter (20 points)



```
W = 3
sobel = nn.Conv2d(1, 2, W, padding=(1,1), bias=False)
T_data = [[[[1., 0., -1.], [2., 0., -2.],[1., 0., -1.]]],
            [[[1., 2., 1.,], [0., 0., 0.], [-1., -2., -1.]]]]
T = torch.tensor(T_data, requires_grad=False)
sobel.weight = torch.nn.Parameter(T, requires_grad=False)
c = sobel(x)
c = torch.sum(torch.square(c), axis=1)
```

Figure 1: Sobel filter code snippet

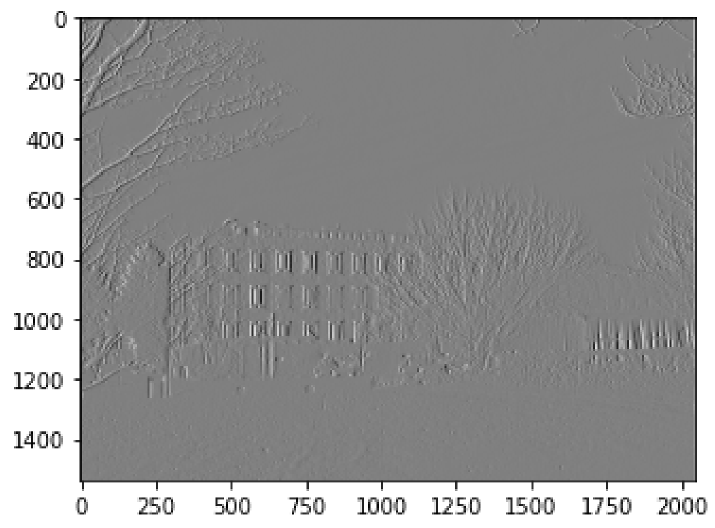


Figure 2: G_x

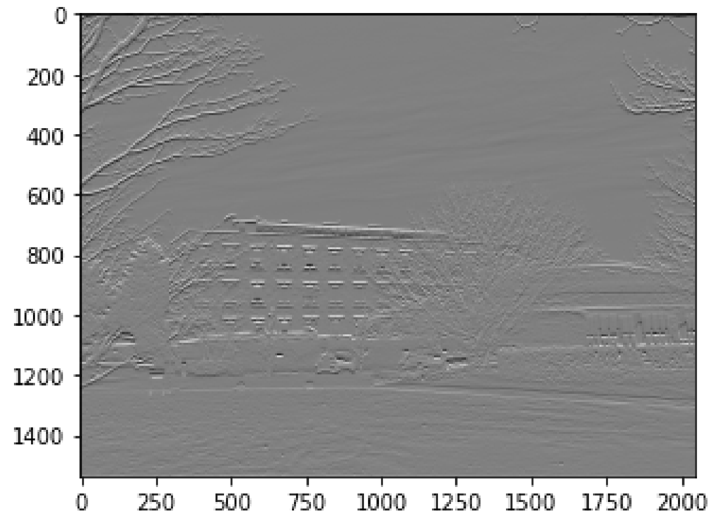


Figure 3: G_y

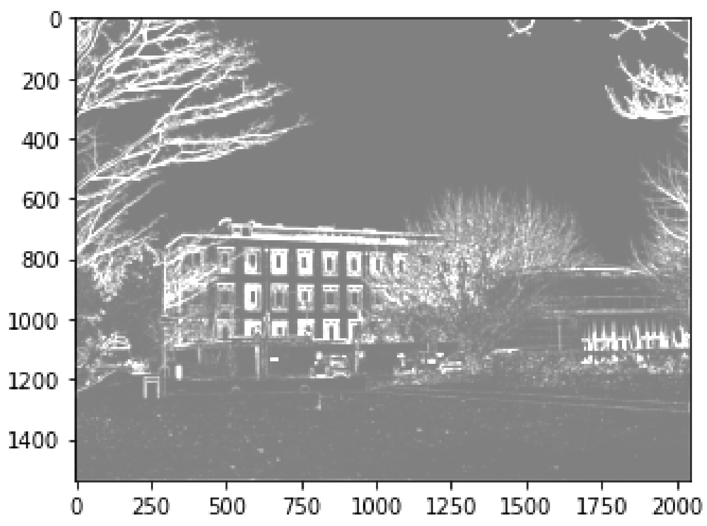
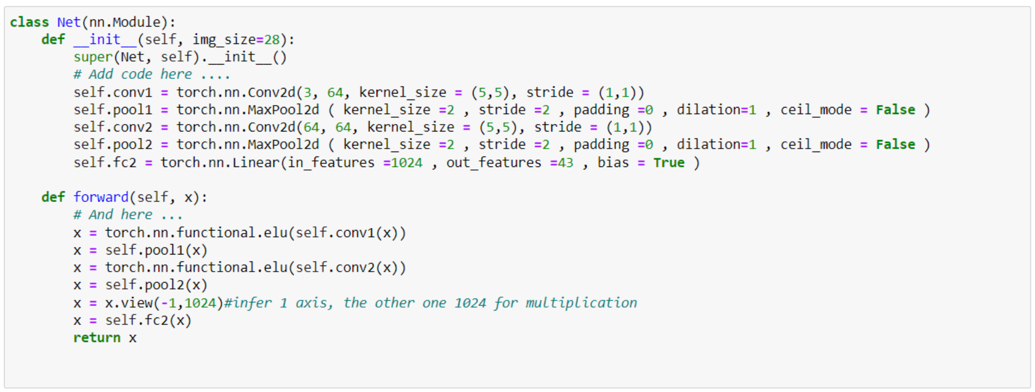


Figure 4: G

Since the kernels are symmetric it doesn't really matter if we apply a cross-correlation or a convolution.
I found it more intuitive to apply cross-correlation, in this way I didn't have to think of flipping the kernels.
the fact that we don't need to compute the flipping operation should lead to faster computation.

1.2 Convolutoinal neural networks(20 points)



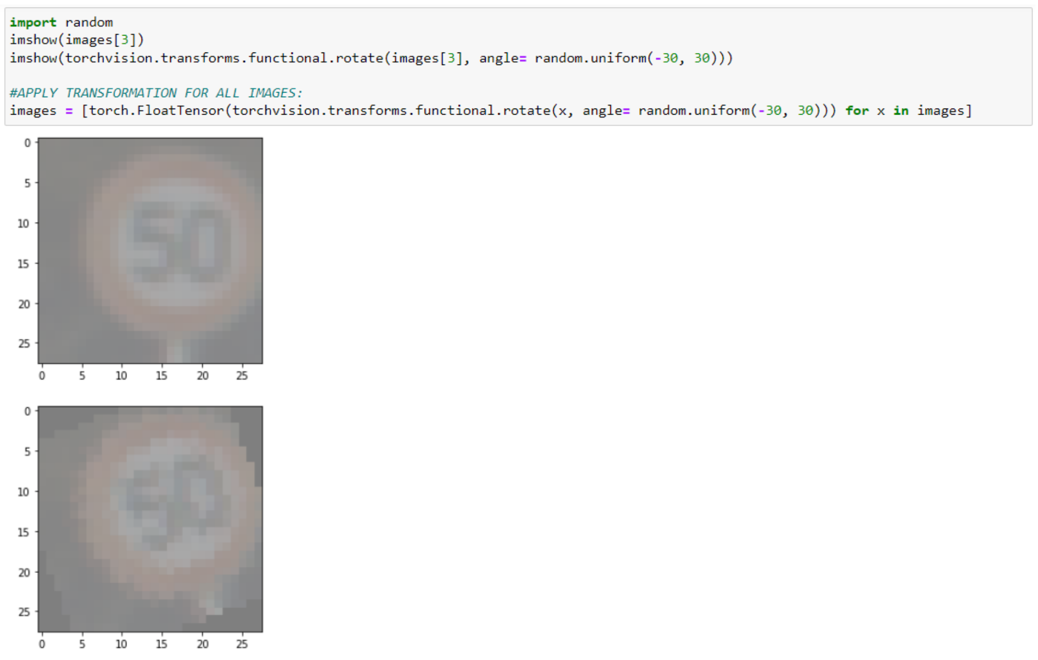
```
class Net(nn.Module):
    def __init__(self, img_size=28):
        super(Net, self).__init__()
        # Add code here ....
        self.conv1 = torch.nn.Conv2d(3, 64, kernel_size = (5,5), stride = (1,1))
        self.pool1 = torch.nn.MaxPool2d ( kernel_size =2 , stride =2 , padding =0 , dilation=1 , ceil_mode = False )
        self.conv2 = torch.nn.Conv2d(64, 64, kernel_size = (5,5), stride = (1,1))
        self.pool2 = torch.nn.MaxPool2d ( kernel_size =2 , stride =2 , padding =0 , dilation=1 , ceil_mode = False )
        self.fc2 = torch.nn.Linear(in_features =1024 , out_features =43 , bias = True )

    def forward(self, x):
        # And here ...
        x = torch.nn.functional.elu(self.conv1(x))
        x = self.pool1(x)
        x = torch.nn.functional.elu(self.conv2(x))
        x = self.pool2(x)
        x = x.view(-1,1024)#infer 1 axis, the other one 1024 for multiplication
        x = self.fc2(x)
        return x
```

Figure 5: Model definition

1.3 Augmentation (10 points)

The transformations applied to the inputs are: changing the background of the image and changing the intensity of the light.
the transformations are conditioned on the labels, because if I rotate an image for example a 9 and I rotate it by 180 degrees, it becomes a 6, so the model should not label it as a 9 anymore, thus, in this case we know that for label 9 we can't apply that transformation (rotation 180 degrees), the same can happen with other examples and other images, so that's why transformations depend on labels.



```python
import random
imshow(images[3])
imshow(torchvision.transforms.functional.rotate(images[3], angle= random.uniform(-30, 30)))

#APPLY TRANSFORMATION FOR ALL IMAGES:
images = [torch.FloatTensor(torchvision.transforms.functional.rotate(x, angle= random.uniform(-30, 30))) for x in images]
```

Figure 6: Augmentation of data

The transformation applied is useful because it might happen that traffic signs are slightly rotated with respect to their original axis.
They might be rotated because they are old traffic signs and need restoration or because they have been vandalised.
I only applied a rotation of 30 degrees, it might be helpful also a slightly higher rotation (like 90 degrees), but a stronger rotation is usually unprobable in real life scenarios so I opted for a slight rotation.

2 Variable Stars (50 points)

2.1 Data understanding and preprocessing (8 points)

1.
frequencies of classes in orginal dataset (class, frequency):
0: 0.08819714656290532, 1: 0.028534370946822308, 2: 0.0012970168612191958, 3: 0.1245136186770428, 4: 0.02204928664072633, 5: 0.0648508430609598, 6: 0.07782101167315175, 7: 0.01297016861219196, 8: 0.03501945525291829, 9: 0.07522697795071336, 10: 0.011673151750972763, 11: 0.027237354085603113, 12: 0.02464332036316472, 13: 0.011673151750972763, 14: 0.027237354085603113, 15: 0.03372243839169909, 16: 0.029831387808041506, 17: 0.03372243839169909, 18: 0.007782101167315175, 19: 0.01297016861219196, 20: 0.0038910505836575876, 21: 0.009079118028534372, 22: 0.10635538261997406, 23: 0.08819714656290532, 24: 0.041504539559014265

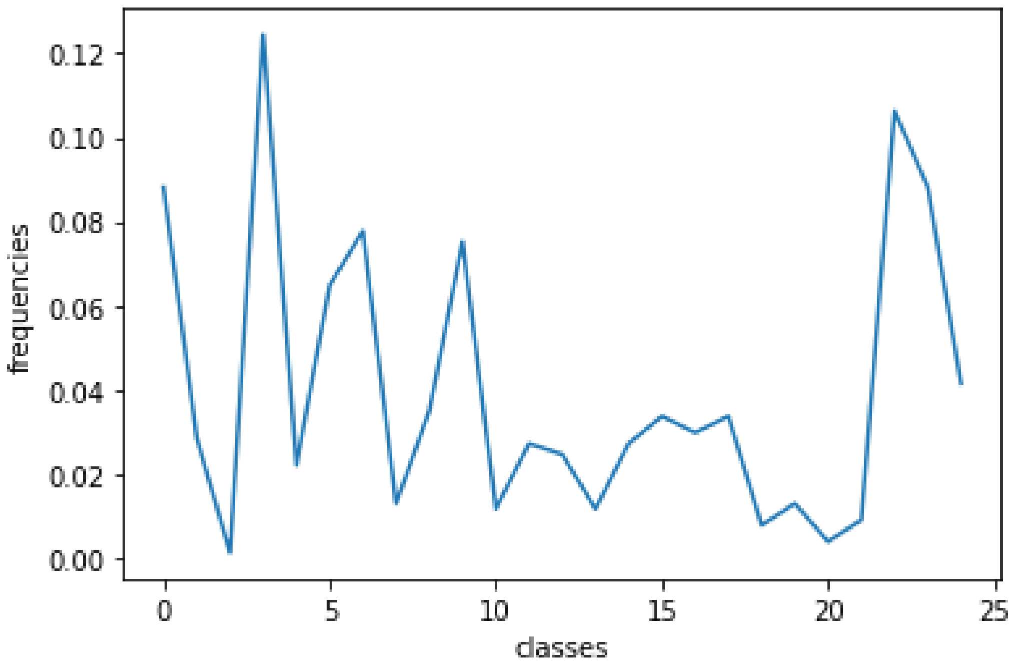


Figure 7: train frequencies

The classes with less than 65 occurences in training set are: [1, 2, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 24]

The remained classes are:[0, 3, 22, 23] with occurencies: [68, 96, 82, 68]

Frequencies of classes in original test set: 0: 0.09857328145265888, 0: 0.09857328145265888, 1: 0.02594033722438392, 2: 0.00648508430609598, 3: 0.12321660181582361, 4: 0.007782101167315175, 5: 0.057068741893644616, 6: 0.08300907911802853, 7: 0.019455252918287938, 8: 0.038910505836575876, 9: 0.07263294422827497, 10: 0.005188067444876783, 11: 0.023346303501945526, 12: 0.01297016861219196, 13: 0.02464332036316472, 14: 0.0311284046692607, 15: 0.03761348897535668, 16: 0.03631647211413749, 17: 0.018158236057068743, 18: 0.010376134889753566, 19: 0.00648508430609598, 20: 0.005188067444876783, 21: 0.007782101167315175, 22: 0.11284046692607004, 23: 0.09987029831387809, 24: 0.03501945525291829

The classes with less than 65 occurences in test set are: [1, 2, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 24]
The remained classes are:[0, 3, 22, 23] with occurencies: [76, 95, 87, 77]

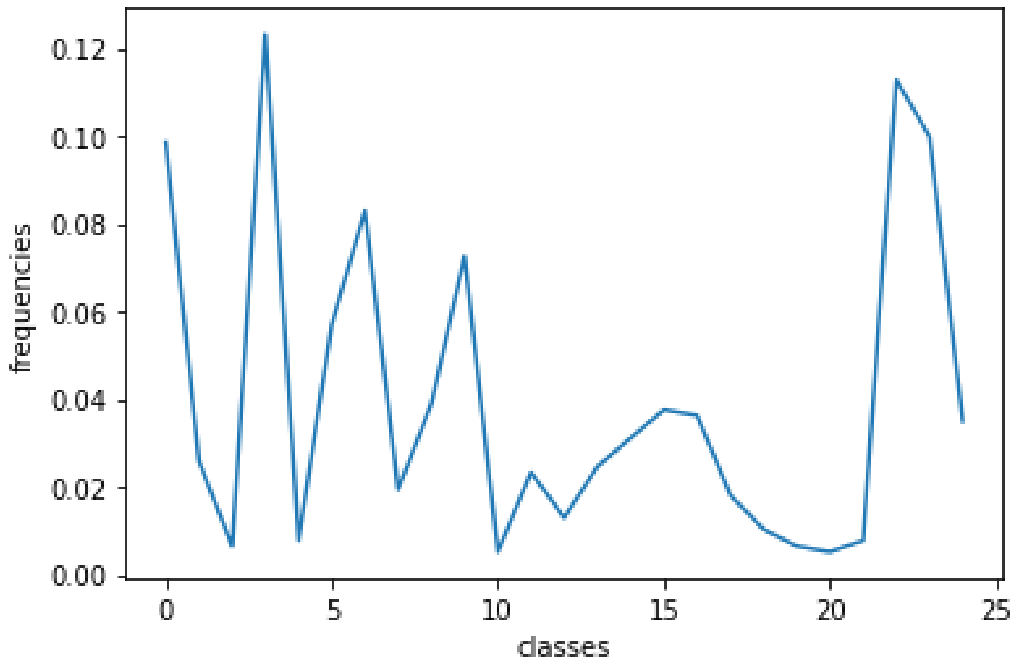
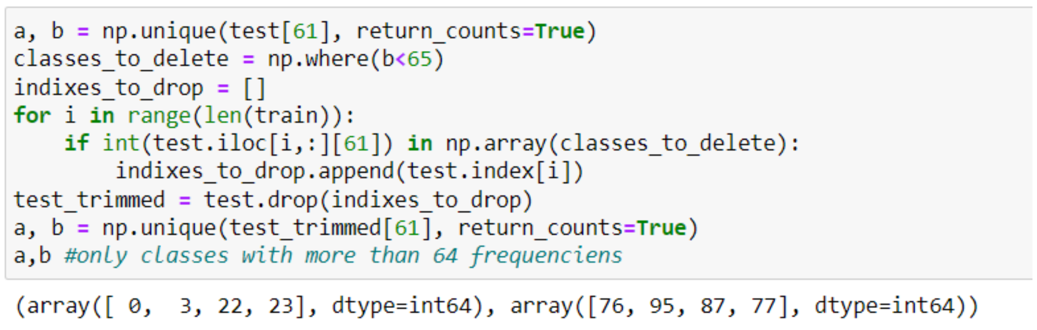


Figure 8: test frequencies

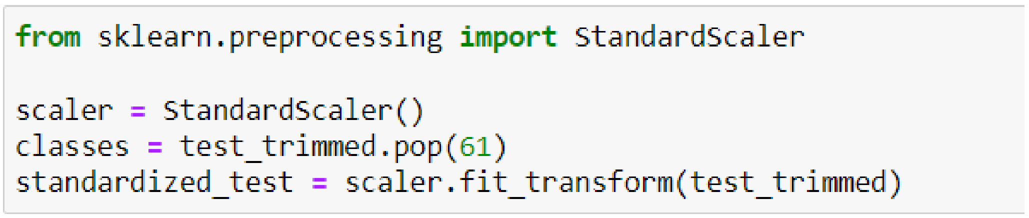
I've provided the code snippets for the test data, but the same applies for the training data



```
a, b = np.unique(test[61], return_counts=True)
classes_to_delete = np.where(b<65)
indixes_to_drop = []
for i in range(len(train)):
    if int(test.iloc[i,:][61]) in np.array(classes_to_delete):
        indixes_to_drop.append(test.index[i])
test_trimmed = test.drop(indixes_to_drop)
a, b = np.unique(test_trimmed[61], return_counts=True)
a,b #only classes with more than 64 frequenciens
```

```
(array([ 0,  3, 22, 23], dtype=int64), array([76, 95, 87, 77], dtype=int64))
```

Figure 9: code snippet for removing classes with less than 65 frequencies



```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
classes = test_trimmed.pop(61)
standardized_test = scaler.fit_transform(test_trimmed)
```

Figure 10: code snippet for standardization of inputs

2.2 Principle component analysis (20 points)

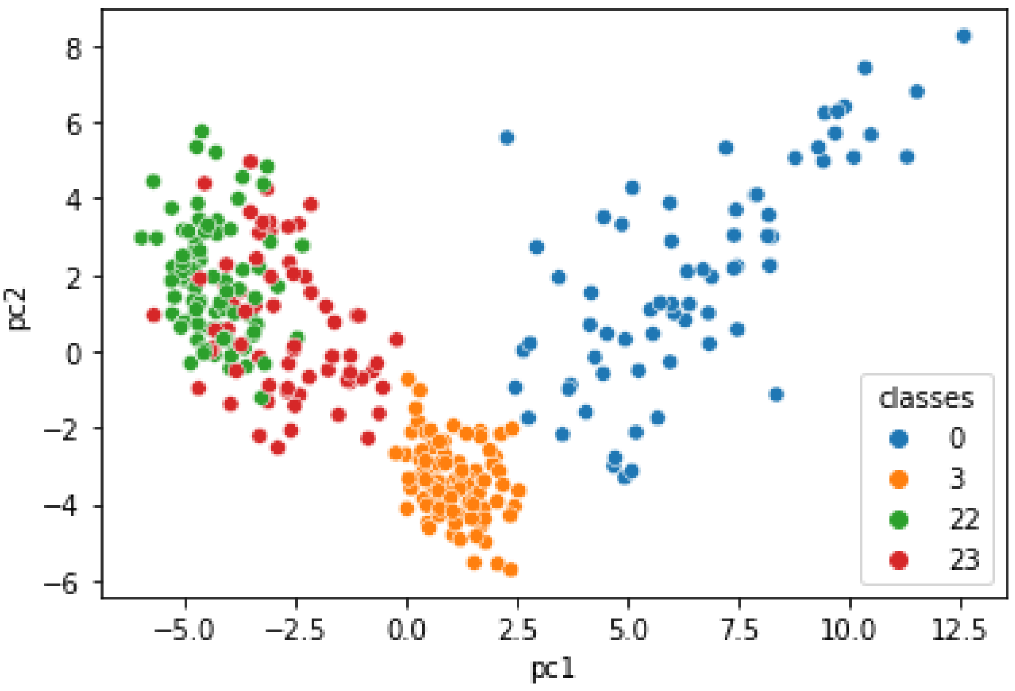


Figure 11: scatterplot after computing pc1 and pc2

2.3 Clustering (22 points)

I use sklearn K-means implementation, in order to perform k-means and k-means++ I changed the init parameter which is 'random' for k-means and 'k-means++' for k-means++.
I ran the algorithms with n_init=1 so the plotted centers are computed after 1 run and they are not the results of the best run after n_init = 10 runs (default)

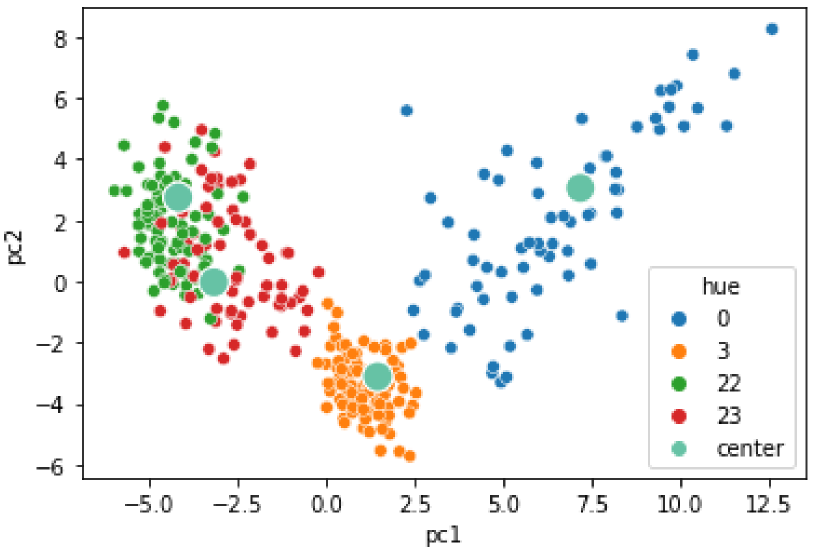


Figure 12: K-means centers

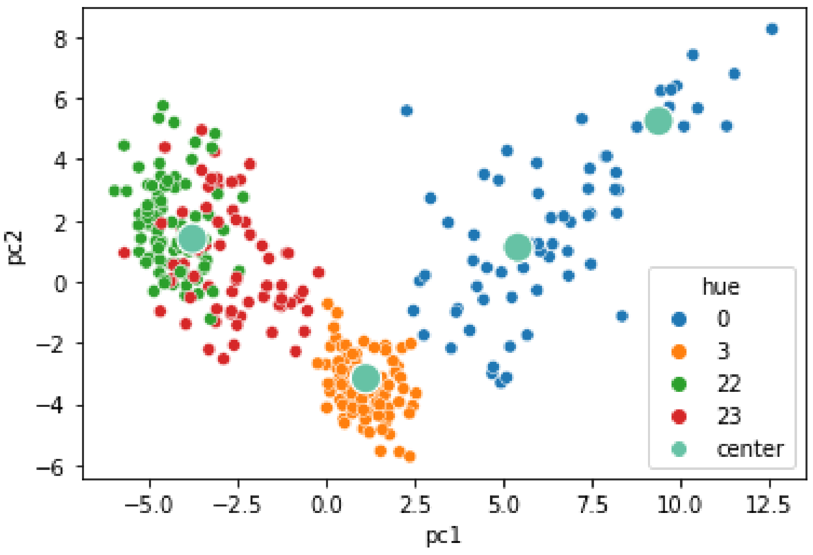


Figure 13: k-means++ centers

The results show that k-means had a good random initialization that converged to a better final result with respect to k-means++.
Probably I would have obtained the same results with the two algorithms if I had taken a larger n_init