Assignment 6

1. Convolutional Neural Network (50 points)

1.1 Sobel filter (20 points)

We implement the Sobel filter. We first define the G_y and G_x matrices as tensors, and combine these into a single tensor, such that we can apply the two feature maps simoultanesously to the image with a single `nn.Conv2d` layer. We obtain the final Sobel filter by taking the square root of the squared feature maps. The code is documented below.

```
## Implement Sobel filter
G_x = torch.tensor([[[[1,0,-1],[2,0,-2],[1,0,-1]]]],dtype=torch.float) # G_x matrix as tensor
G_y = torch.tensor([[[[1,2,1],[0,0,0],[-1,-2,-1]]]],dtype=torch.float) # G_y matrix as tensor

sobel_comb = torch.cat((G_x,G_y)) # Combine Sobel matrices into single tensor
convG = nn.Conv2d(1, 2, kernel_size=(3, 3), bias=False)
#1 input channel, 2 output channel corresponding to G_x and G_y

convG.weight = torch.nn.Parameter(sobolev_comb, requires_grad = False) # Set weights

c_G = convG(x) # Convolve the images

sobel = torch.sqrt(torch.sum(torch.square(c_G), (0,1))) # Obtain final Sobel filter
```

We can now plot the feature maps and the Sobel filter

Figure 1: G_x

Figure 2: G_y

Figure 3: G

What we have done is actually not convolution, but rather cross-correlation, due to how Pytorch has implemented convolutional layers. Cross-correlation is simply convolution with the kernel flipped 180 degrees. If we had done convolution instead, figure 3 would therefore have been flipped 180 degrees, such that the DIKU sign was on its head and mirrored in the vertical axis of the picture.

1.2 Convolutional neural network (20 points)

Below we define the convolutional neural network as described in the assignment

```
class Net(nn.Module):
    def __init__(self, img_size=28):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 64, kernel_size = (5,5), stride =(1,1))
```

```
        # 3 channels, 64 feature maps, 5x5 kernel
        self.pool1 = nn.MaxPool2d(kernel_size =2 , stride =2 , padding =0 , dilation =1 ,
         ceil_mode = False ) #2x2 max-pool
        self.conv2 = nn.Conv2d(64 , 64 , kernel_size =(5 , 5) , stride =(1 , 1) )
         # 64 feature maps, 5x5 kernel
        self.pool2 = nn.MaxPool2d ( kernel_size =2 , stride =2 , padding =0 , dilation =1 ,
        ceil_mode = False ) # 2x2 max-pool
        self.fc2 = nn.Linear(in_features =1024 , out_features =43 , bias = True ) # Linear output

    def forward(self, x):
        x = self.conv1(x) # Convolution
        x = F.elu(x) # Elu activation
        x = self.pool1(x) # max-pool
        x = self.conv2(x) # Convolution
        x = F.elu(x) # elu activation
        x = self.pool2(x) # max-pool
        x = self.fc2(torch.flatten(x,1)) # flatten before linear outputlayer
        return x
```

1.3 Augmentation (10 points)

Looking at the code for augmentation, we see that the following augmentation procedures are used:

- Random rotation of the images, between -5 and 5 degrees using 'RandomAffine'.
- Random cropping of images using 'RandomCrop'.
- Random changes to brightness, contrast and other picture quality features using 'ColorJitter'.
- Random horizontal flips (at probability 0.5) using 'RandomHorizontalFlip' on images with certain labels.

The Random horizontal flips are conditioned on the label. If we look at the images corresponding to these labels, we notice that they are all horizontally symmetric. The data can therefore be augmented by flipping the images horizontally at random, because the flipped sign looks, in principle, identical to the original sign. This augmentation is nonsensical (if we assume that traffic signs in the wild being flipped, is very rare) to apply to, say, a STOP-sign, because this sign is most certainly not horizontally symmetric. \

A reasonable additional augmentation technique to add to the images, could be, to at random slightly change the perspective of the image. This might be reasonable, when considering that cars (who are presumably the entities using the algorithm) most likely will have cameras mounted on at different positions, and will therefore see traffic signs from different perspectives. In addition traffic signs in real life also becomes bent by wind and weather, which also gives a different perspective on them from the road. We can implement this augmentation by adding the following piece of code to the augmentation part of the code

```
image = transforms.RandomPerspective(distortion_scale = 0.05, p = 0.5)(image)
```

Note that we are rather conservative with our `distortion_scale` parameter, as to not totally obscure the image.

2. Variable Stars (50 point)

2.1 Data understanding and prepocessing (8 point)

Class frequencies

We report the frequencies of the classes in the table below

Table 1: Label and class frequency for starts

Label	Frequency
0	0.0882
1	0.0285
2	0.0013
3	0.1245
4	0.0220
5	0.0649
6	0.0778
7	0.0130
8	0.0350
9	0.0752
10	0.0117
11	0.0272
12	0.0246
13	0.0117
14	0.0272
15	0.0337
16	0.0298
17	0.0337
18	0.0078
19	0.0130
20	0.0039
21	0.0091
22	0.1064
23	0.0882
24	0.0415

Removing data points

We remove the data points belonging to classes with less than 65 training examples. The code is documented below

```
#Clean up datasets
train = np.delete(train, np.where(True==np.isin(train[:,61],
np.where(train_count[1]<65)[0])), axis = 0)
# np.where(train_count[1]<65)[0] returns the labels that have fewer than 65 observations
# np.isin(train[:,61], np.where(...)) returns true for each label if the label has fewer than 65
# np.where(True== ...) gets the indices of the rows with labels with fewer than 65 observations
#np.delete(..., axis = 0) deletes the corresponding rows
test = np.delete(test, np.where(True==np.isin(test[:,61],
np.where(train_count[1]<65)[0])), axis = 0)
```

After this cleaning of the data, the training data set has 314 data points and the test dataset has 335 datapoints. The labels left in the datasets are 0, 3, 22 and 23, corresponding to the classes Mira, Classical Cep, Beta Persei and Beta Lyrae.

Rescaling to unit variance and zero mean

We rescale each component of the features to unit variance and zero mean. This is done by subtracting the mean of every feature, and afterwards dividing by the standard deviation of each feature. The code is shown below

```
train_norm = ((train - train.mean(axis=0))/train.std(axis=0))
```

2.2 Principal component analysis (20 point)

We do PCA on the normalised training data, using the first two principal components, utilising the `sklearn.decomposition` class `PCA`. We visualise the normalized training data by projecting the normalized training data onto the first two principal components.

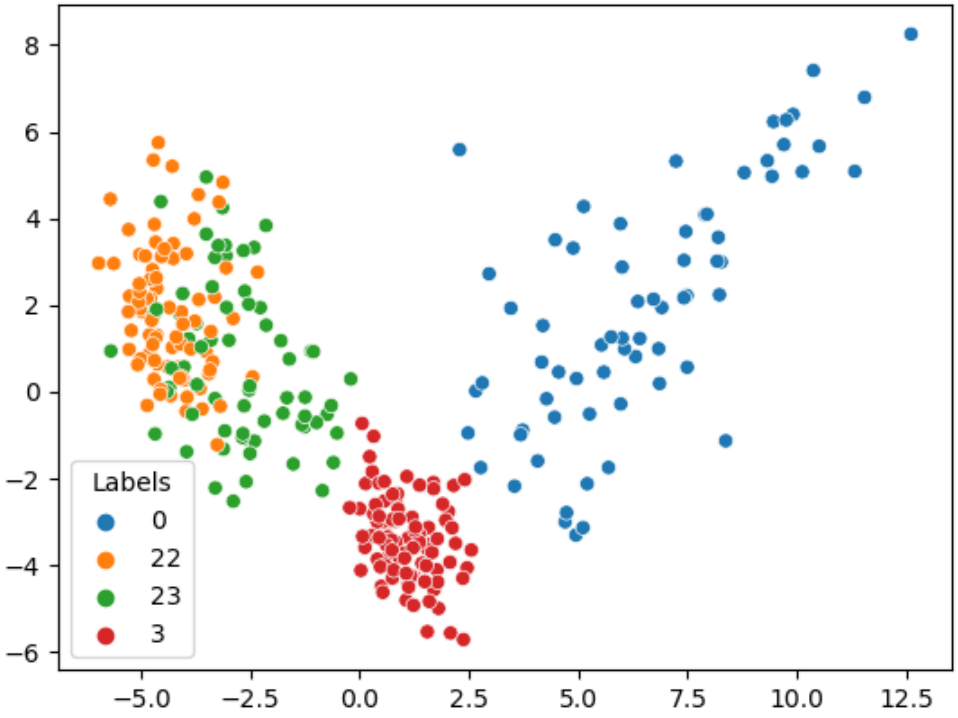

Figure 4: Normalized training data projected onto 2 principal components

2.3 Clustering (22 points)

We perform 4-means clustering and 4-means++ clustering on the normalised training data, using the `KMeans` class from `sklearn.cluster`, with `n_clusters` set to 4. This class has two methods for initialisation either `random` which corresponds to random initial clusters which is the classic k-means algorithm, or `k-means++` which corresponds to the k-means++ algorithm - note that in `sklearn` implementation the initial clusters are still random if k-means++ is the initialsation. Further the class is implemented with a paramater `n_init`. This parameter specifies how many time to perform the clustering, each time with a new seeding. The output of the algorithm is then the output of the clustering on the initialisation which minimizes the k-means objective function. We set this parameter to 1 as the exercise asks us to perform a *single* clustering. We keep the rest of the parameters at their default values. We plot the centroids projected on the 2 principal components of the normalised training set

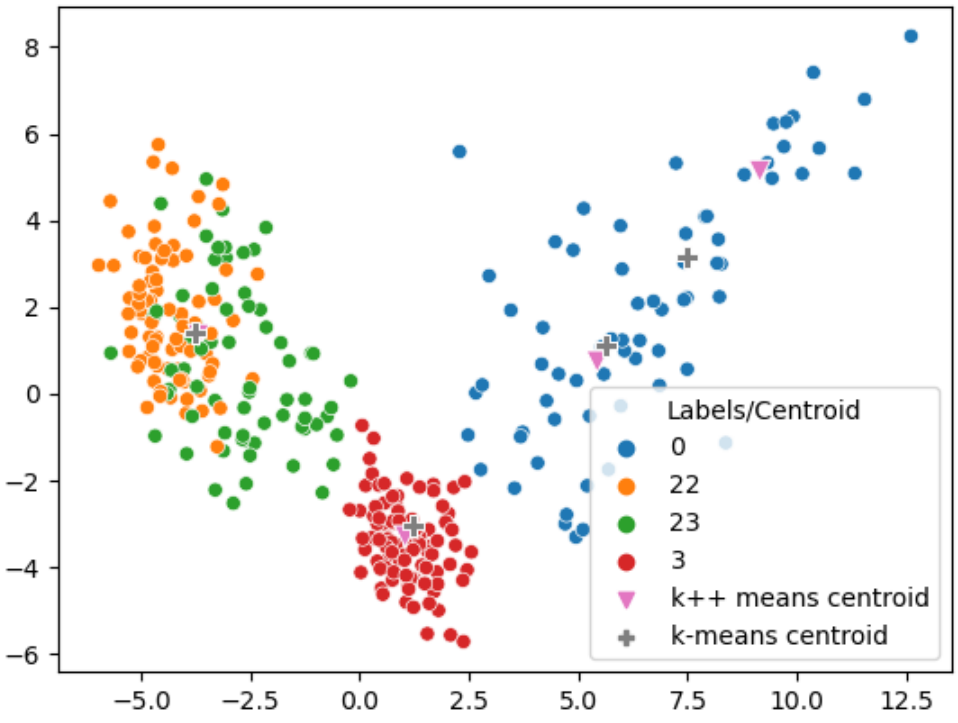

Figure 5: Centroids projected onto 2 principal components

Both the algorithms seem to capture the the concentration of points around the labels 3 and 22/23 on the projected data. They also seem to agree on a clustering around the 0-label. We also note that the centroids are output along (close to) the two principal components. The output of the two algorithms is in this case the same, however as both are random by nature, we would need to run the algorithms numerous times to conclude something about the stability and precision of the algorithms compared to each other.