Weather Classification

In this task, I applied a simple Convolutional Neural Network (CNN) on the dataset to classify weather images into 4 categories of cloudy, rain, shine, and sunrise. The dataset contains 836 images of different sizes and channels.

The CNN Model has 4 convolutional layers. RelU activation function is applied right after each convolution layer and then the output is max pooled. I used 3 fully connected layers at the end of the CNN. Adding the fully-connected layers helps learn non-linear combinations of the high-level features outputted by the convolutional layers. Also to avoid overfitting of the model I used batchNormalization on convolutional layers and Dropouts on the fully connected layers.

Data exploration:

The dataset was uncleaned. Other than RGB (3-channels) images there is(are) at least a grayscale image (1-channel) and a 4-channels image. I also find images are used more than once with were either resized or flipped.

The image datases is devided to training data and validation data split (80% - 20%). The images are resized to 64x64 images. The larger images would make the training slower, and small images would reduce the accuracy of the training.

Code

I have included small explanations for each step of the code. I have run the code on my cpu. General steps are as follows:

- Read all images
- Clean and resize images
- Split images to train/test sets
- Define a Convolutional Neural Network
- Train the model and evaluate the trained model
- Use the model

The requirements for running the code:

- Jupyter notebook
- pytorch

- torchvision
- torchsummary
- matplotlib
- skimage
- tqdm

In [28]:

```
import glob
import numpy as np
import os, sys

# read the image list
image_list = glob.glob('./data/*.jpg')
```

In [29]:

```
print(len(image_list))
```

836

In [11]:

```
import tqdm
from skimage import io, transform
import torch
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import Dataset, DataLoader
```

Function resize_img() resize the original images to 64x64 images and also checks for images of different channles and adjust them to a 3-channel image.

In [ ]:

```
# resize (64x64) and save images
oldpath = "./data/"
path = "./resized/"

def resize_img(image_list):
    for img_name in tqdm.tqdm(image_list):
        image = io.imread(img_name)
        image_resized = transform.resize(image, (64, 64),anti_aliasing=True)
        if len(image_resized.shape) == 2:
            image_resized = np.tile(np.expand_dims(image_resized, axis=-1), (1,1,3))
        assert len(image_resized.shape) == 3, f'{img_name} - {image_resized.shape}'
```

```
        image_resized = image_resized[:, :, :3]
        assert image_resized.shape == (64, 64, 3), f'{img_name} - {image_resized.shape}'
        basename = os.path.basename(img_name)
        image_resized = (image_resized * 255).astype(np.uint8)
        io.imsave(os.path.join(path,basename).replace('.jpg', '.png') , image_resized)

resize_img(image_list)
```

In [12]:

```
#read from resized images
resized_image_list = glob.glob('./resized/*.png')
```

A class made for dataset. A class of dataset makes it easier for accessing images and their labels.

In [13]:

```
class weatherDataset(Dataset):
    """weather Dataset."""

    def __init__(self, image_list, image_labels):
        self.images = image_list
        self.labels = image_labels

    def __len__(self):
        return len(self.images)

    def __getitem__(self, idx):
        if torch.is_tensor(idx):
            idx = idx.tolist()

        img_name = self.images[idx]
        image = io.imread(img_name)
        label = self.labels[idx]
        image = np.transpose(image, (2,0,1)) # transpose the images to macth the tensor format
        image = image[:3,:,:]
        assert image.shape == (3, 64, 64), f'{img_name} - {image.shape}'

        return image,label
```

I used 80/20 split for training and testing.

In [14]:

```
train_size = int(0.8 * len(resized_image_list))
test_size = len(resized_image_list) - train_size
```

```
train_imageList, test_imageList = torch.utils.data.random_split(resized_image_list, [train_size, test_size])
```

In [ ]:

```
# a sinple function to map the images with their labels
def map_filename_to_label(filename):
    if 'cloudy' in filename:
        return 0
    elif 'rain' in filename:
        return 1
    elif 'shine' in filename:
        return 2
    elif 'sunrise' in filename:
        return 3

#
test_labels = list(map(map_filename_to_label, test_imageList))
train_labels = list(map(map_filename_to_label, train_imageList))
```

In [15]:

```
# Two datasets are defined for training and test images.
trainData = weatherDataset(train_imageList, train_labels)
testData = weatherDataset(test_imageList, test_labels)
```

Pytorch function DataLoader used to load datasets for traing and testing. I set the batch_size = 4 (larger amounts would cause runtime error).

In [16]:

```
trainLoader = DataLoader(trainData, batch_size=4, shuffle=True, num_workers=0,
           drop_last=True, prefetch_factor=2)
testLoader = DataLoader(testData, batch_size=4, shuffle=False, num_workers=0,
           drop_last=True, prefetch_factor=2)
```

Here you see some example images from training set.

In [17]:

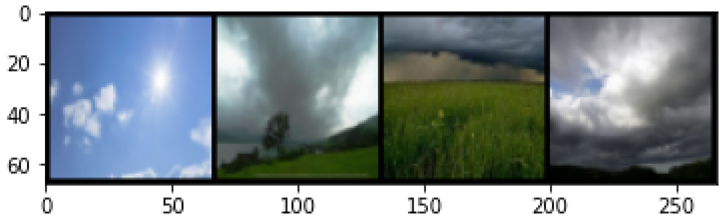
```
import matplotlib.pyplot as plt

# functions to show an image

batch_size = 4
def imshow(img):
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()
```

```python
# get some random training images
dataiter = iter(trainLoader)
images, labels = dataiter.next()

# show images
imshow(torchvision.utils.make_grid(images))
# print labels
print(' '.join('%5s' % train_imageList[labels[j]] for j in range(batch_size)))
```



```
./resized\rain103.png ./resized\cloudy140.png ./resized\cloudy140.png ./resized\cloudy140.png
```

Here I defined a CNN with 4 conv2d layers and 3 fully connected layers.

In [18]:

```python
#Define a Convolutional Neural Network

import torch
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, 3 , padding =1 , bias=False)
        self.conv2_bn1 = nn.BatchNorm2d(16)
        self.conv2 = nn.Conv2d(16, 32, 3 , padding =1 , bias=False)
        self.conv2_bn2 = nn.BatchNorm2d(32)
        self.conv3 = nn.Conv2d(32, 64, 3 , padding =1 , bias=False)
        self.conv2_bn3 = nn.BatchNorm2d(64)
        self.conv4 = nn.Conv2d(64, 128, 3 , padding =1 , bias=False)
        self.conv2_bn4 = nn.BatchNorm2d(128)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(128 *4 *4, 64)
```

```python
        self.fc1_do = nn.Dropout(p = 0.2)
        self.fc2 = nn.Linear(64, 32)
        self.fc2_do = nn.Dropout(p = 0.2)
        self.fc3 = nn.Linear(32, 4)

    def forward(self, x):
        x = self.conv1(x) # 3x64x64 -> 16x64x64
        x = self.pool(F.relu(self.conv2_bn1(x))) # 16x64x64 -> 16x32x32
        x = self.conv2(x) # 16x32x32 -> 32x32x32
        x = self.pool(F.relu(self.conv2_bn2(x))) #  32x32x32 ->  32x16x16
        x = self.conv3(x) #  32x16x16 ->  64x16x16
        x = self.pool(F.relu(self.conv2_bn3(x))) #  64x16x16 ->  64x8x8
        x = self.conv4(x) #  64x8x8 ->  128x8x8
        x = self.pool(F.relu(self.conv2_bn4(x))) #  128x8x8 ->  128x4x4
        x = torch.flatten(x, 1) # 512
        x = F.relu(self.fc1_do(self.fc1(x))) # 120
        x = F.relu(self.fc2_do(self.fc2(x))) # 84
        x = self.fc3(x) # 4
        x = nn.Sigmoid()(x)
        return x
```

In [19]:

```python
# Use torchsummary to visualize the model
from torchsummary import summary
net = Net()
summary(net, input_size=(3, 64, 64))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 16, 64, 64]             432
       BatchNorm2d-2           [-1, 16, 64, 64]              32
         MaxPool2d-3           [-1, 16, 32, 32]               0
            Conv2d-4           [-1, 32, 32, 32]           4,608
       BatchNorm2d-5           [-1, 32, 32, 32]              64
         MaxPool2d-6           [-1, 32, 16, 16]               0
            Conv2d-7           [-1, 64, 16, 16]          18,432
       BatchNorm2d-8           [-1, 64, 16, 16]             128
         MaxPool2d-9             [-1, 64, 8, 8]               0
           Conv2d-10            [-1, 128, 8, 8]          73,728
      BatchNorm2d-11            [-1, 128, 8, 8]             256
        MaxPool2d-12            [-1, 128, 4, 4]               0
           Linear-13                   [-1, 64]         131,136
```

```
         Dropout-14                 [-1, 64]               0
          Linear-15                 [-1, 32]           2,080
         Dropout-16                 [-1, 32]               0
          Linear-17                  [-1, 4]             132
================================================================
Total params: 231,028
Trainable params: 231,028
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.05
Forward/backward pass size (MB): 2.11
Params size (MB): 0.88
Estimated Total Size (MB): 3.04
----------------------------------------------------------------
```

Optimizer

I usd torch.optim to construct an optimizer object. It will hold the current state and will update the parameters based on the computed gradients. I used the stochastic gradient descent method, however, Adam is another method that could be used.

I used cross entropy loss as it's commonly used in classification tasks. It measures the performance of a classification model whose output is a probability value between 0 and 1.

In [20]:

```
import torch.optim as optim
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

In [21]:

```
criterion = nn.CrossEntropyLoss()
```

In [22]:

```
#reset network
for layer in net.children():
   if hasattr(layer, 'reset_parameters'):
       layer.reset_parameters()
```

Training the model

Here, I trained the model. To see accuracy and loss values, the model being tested in each epoch after being trained. The best model with lowest validation loss will be saved in the directory.

In [23]:

```
train_losses=[]
train_accu=[]
eval_losses=[]
eval_accu=[]
best_val_loss = 100000

for epoch in range(60):  # loop over the dataset multiple times
    correct_tr = 0
    total_tr = 0
    running_loss_tr = 0.0

    net.train() # traing mode
    for i, data in enumerate(trainLoader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs.float() )
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # the class with the highest energy is what we choose as prediction
        _, predicted = torch.max(outputs.data, 1)
        total_tr += labels.size(0)
        correct_tr += (predicted == labels).sum().item()

        # sum of losses
        running_loss_tr += loss.item()


    net.eval() # traing mode
    correct_val = 0
    total_val = 0
    running_loss_test = 0.0

    # since we're not training, we don't need to calculate the gradients for our outputs
    with torch.no_grad():
        for data in testLoader:
```

```
            images, labels = data
            # calculate outputs by running images through the network
            outputs = net(images.float())
            loss = criterion(outputs, labels)
            # the class with the highest energy is what we choose as prediction
            _, predicted = torch.max(outputs.data, 1)
            total_val += labels.size(0)
            correct_val += (predicted == labels).sum().item()

            # sum of losses
            running_loss_test += loss.item()

    # print Accuracy and losses for train set in each epoch
    train_loss=running_loss_tr/len(trainLoader)
    tr_accu = 100 * correct_tr / total_tr

    train_losses.append(train_loss)
    train_accu.append(tr_accu)
    print('Train Loss: %.3f | Accuracy: %.3f'%(train_loss,tr_accu))

    # print Accuracy and losses for test set in each epoch
    test_loss=running_loss_test/len(testLoader)
    test_accu = 100 * correct_val / total_val

    eval_losses.append(test_loss)
    eval_accu.append(test_accu)

    print('Test Loss: %.3f | Accuracy: %.3f'%(test_loss,test_accu))

    #stopping criteria when reached to lowest validation loss
        if test_loss < best_val_loss:
        best_val_loss = test_loss
        tPATH = f'./weather_net_{epoch}_{int(test_loss)}_{int(test_accu)}.pth'
        torch.save(net.state_dict(), tPATH)
    if test_loss >  best_val_loss + 1:
        print('early stop')
        break

print('Finished Training')
```

```
Train Loss: 1.322 | Accuracy: 36.527
Test Loss: 1.278 | Accuracy: 45.833
Train Loss: 1.249 | Accuracy: 38.174
```

```
Test Loss: 1.192 | Accuracy: 45.238
Train Loss: 1.186 | Accuracy: 44.311
Test Loss: 1.126 | Accuracy: 45.833
Train Loss: 1.136 | Accuracy: 44.910
Test Loss: 1.100 | Accuracy: 49.405
Train Loss: 1.130 | Accuracy: 52.395
Test Loss: 1.067 | Accuracy: 57.143
Train Loss: 1.094 | Accuracy: 53.443
Test Loss: 1.058 | Accuracy: 63.095
Train Loss: 1.077 | Accuracy: 58.234
Test Loss: 1.110 | Accuracy: 51.190
Train Loss: 1.053 | Accuracy: 64.820
Test Loss: 0.990 | Accuracy: 80.357
Train Loss: 1.032 | Accuracy: 73.503
Test Loss: 0.937 | Accuracy: 83.929
Train Loss: 1.000 | Accuracy: 75.749
Test Loss: 0.919 | Accuracy: 82.738
Train Loss: 0.969 | Accuracy: 76.198
Test Loss: 0.886 | Accuracy: 85.714
Train Loss: 0.939 | Accuracy: 81.138
Test Loss: 0.958 | Accuracy: 75.595
Train Loss: 0.915 | Accuracy: 84.281
Test Loss: 0.863 | Accuracy: 89.286
Train Loss: 0.915 | Accuracy: 82.485
Test Loss: 0.838 | Accuracy: 90.476
Train Loss: 0.899 | Accuracy: 83.832
Test Loss: 0.869 | Accuracy: 87.500
Train Loss: 0.885 | Accuracy: 86.377
Test Loss: 0.901 | Accuracy: 80.952
Train Loss: 0.907 | Accuracy: 82.036
Test Loss: 0.877 | Accuracy: 86.905
Train Loss: 0.878 | Accuracy: 86.527
Test Loss: 0.847 | Accuracy: 90.476
Train Loss: 0.875 | Accuracy: 85.928
Test Loss: 0.861 | Accuracy: 86.905
Train Loss: 0.859 | Accuracy: 89.072
Test Loss: 0.871 | Accuracy: 88.095
Train Loss: 0.854 | Accuracy: 89.072
Test Loss: 0.822 | Accuracy: 91.071
Train Loss: 0.840 | Accuracy: 90.868
Test Loss: 0.819 | Accuracy: 92.262
Train Loss: 0.846 | Accuracy: 89.970
Test Loss: 0.851 | Accuracy: 89.881
Train Loss: 0.840 | Accuracy: 89.521
```

```
Test Loss: 0.823 | Accuracy: 91.667
Train Loss: 0.829 | Accuracy: 92.814
Test Loss: 0.861 | Accuracy: 88.095
Train Loss: 0.819 | Accuracy: 92.814
Test Loss: 0.810 | Accuracy: 93.452
Train Loss: 0.825 | Accuracy: 92.066
Test Loss: 0.814 | Accuracy: 92.857
Train Loss: 0.799 | Accuracy: 94.611
Test Loss: 0.806 | Accuracy: 94.048
Train Loss: 0.807 | Accuracy: 94.461
Test Loss: 0.871 | Accuracy: 86.310
Train Loss: 0.811 | Accuracy: 93.413
Test Loss: 0.863 | Accuracy: 88.095
Train Loss: 0.801 | Accuracy: 95.210
Test Loss: 0.791 | Accuracy: 96.429
Train Loss: 0.797 | Accuracy: 95.808
Test Loss: 0.821 | Accuracy: 92.857
Train Loss: 0.797 | Accuracy: 94.611
Test Loss: 0.851 | Accuracy: 89.286
Train Loss: 0.798 | Accuracy: 95.210
Test Loss: 0.845 | Accuracy: 90.476
Train Loss: 0.788 | Accuracy: 95.958
Test Loss: 0.815 | Accuracy: 92.262
Train Loss: 0.782 | Accuracy: 96.707
Test Loss: 0.798 | Accuracy: 94.643
Train Loss: 0.789 | Accuracy: 96.108
Test Loss: 0.801 | Accuracy: 94.643
Train Loss: 0.779 | Accuracy: 96.856
Test Loss: 0.818 | Accuracy: 92.857
Train Loss: 0.775 | Accuracy: 97.305
Test Loss: 0.802 | Accuracy: 93.452
Train Loss: 0.775 | Accuracy: 97.305
Test Loss: 0.817 | Accuracy: 92.262
Train Loss: 0.770 | Accuracy: 97.904
Test Loss: 0.803 | Accuracy: 94.643
Train Loss: 0.782 | Accuracy: 96.407
Test Loss: 0.823 | Accuracy: 91.667
Train Loss: 0.771 | Accuracy: 97.605
Test Loss: 0.811 | Accuracy: 93.452
Train Loss: 0.763 | Accuracy: 98.952
Test Loss: 0.800 | Accuracy: 94.048
Train Loss: 0.765 | Accuracy: 98.353
Test Loss: 0.805 | Accuracy: 92.262
Train Loss: 0.768 | Accuracy: 98.054
```

```
Test Loss: 0.841 | Accuracy: 89.286
Train Loss: 0.761 | Accuracy: 99.251
Test Loss: 0.819 | Accuracy: 91.667
Train Loss: 0.763 | Accuracy: 98.503
Test Loss: 0.846 | Accuracy: 87.500
Train Loss: 0.768 | Accuracy: 98.204
Test Loss: 0.829 | Accuracy: 90.476
Train Loss: 0.762 | Accuracy: 98.204
Test Loss: 0.813 | Accuracy: 92.857
Train Loss: 0.763 | Accuracy: 98.204
Test Loss: 0.799 | Accuracy: 94.643
Train Loss: 0.765 | Accuracy: 98.204
Test Loss: 0.807 | Accuracy: 93.452
Train Loss: 0.760 | Accuracy: 98.802
Test Loss: 0.793 | Accuracy: 94.048
Train Loss: 0.760 | Accuracy: 98.653
Test Loss: 0.798 | Accuracy: 94.643
Train Loss: 0.765 | Accuracy: 98.204
Test Loss: 1.072 | Accuracy: 64.286
Train Loss: 0.770 | Accuracy: 97.455
Test Loss: 0.813 | Accuracy: 92.262
Train Loss: 0.767 | Accuracy: 97.904
Test Loss: 0.807 | Accuracy: 93.452
Train Loss: 0.764 | Accuracy: 98.353
Test Loss: 0.805 | Accuracy: 94.048
Train Loss: 0.754 | Accuracy: 99.551
Test Loss: 0.826 | Accuracy: 90.476
Train Loss: 0.759 | Accuracy: 98.952
Test Loss: 0.809 | Accuracy: 93.452
Finished Training
```

Accuracy and loss

The first plot shows the accuracy of the training set vs test set, and second plot shows the losses of the training set vs test set. As in the plots we can see the accuracy stopes around 96%.

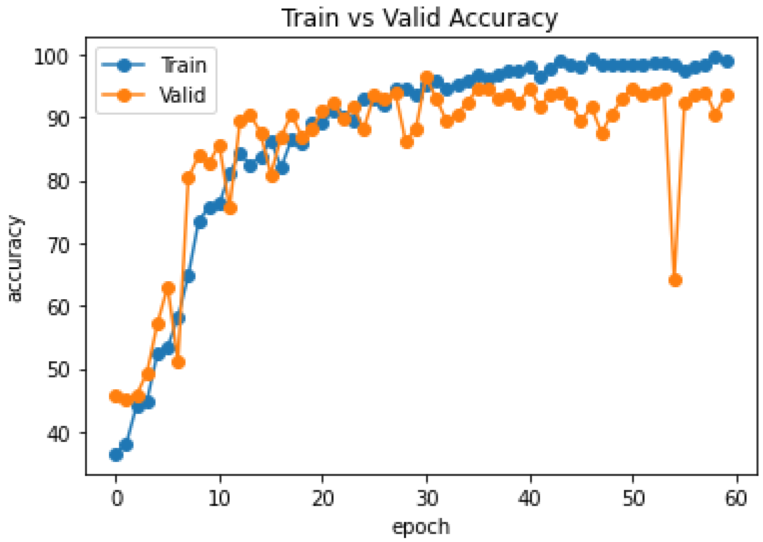
In [24]:

```
#plot accuracy

plt.plot(train_accu,'-o')
plt.plot(eval_accu,'-o')
plt.xlabel('epoch')
plt.ylabel('accuracy')
```

```python
plt.legend(['Train','Valid'])
plt.title('Train vs Valid Accuracy')

plt.show()
```

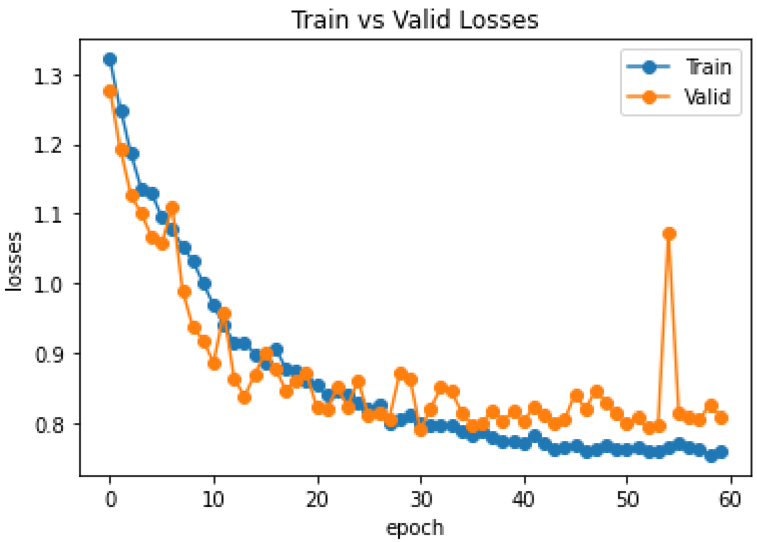


In [25]:

```python
#plot losses

plt.plot(train_losses,'-o')
plt.plot(eval_losses,'-o')
plt.xlabel('epoch')
plt.ylabel('losses')
plt.legend(['Train','Valid'])
plt.title('Train vs Valid Losses')

plt.show()
```



Use the Model

For prediction, we load the best model and run the network to test the test_dataset. The model was able to predict the test images with 96% accuracy. Below you can see 4 example images with the predictions and the grand truth labels. Where {0: cloudy, 1: rain, 2: shine, 3: sunrise}.

In [31]:

```
# Load the best model
Best_Model_Path = './weather_net_30_0_96.pth'
net.load_state_dict(torch.load(Best_Model_Path))
```

Out[31]:

```
<All keys matched successfully>
```

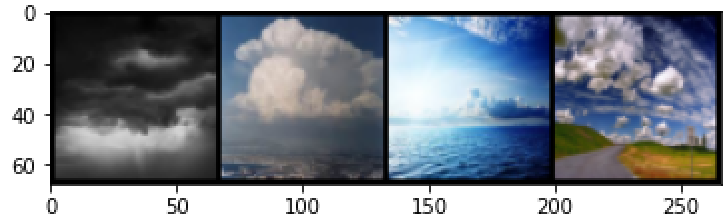
In [32]:

```
dataiter = iter(testLoader)
images, labels = dataiter.next()

outputs = net(images.float())
_, predicted = torch.max(outputs, 1)

print(f'Predicted: {str(predicted)}')
imshow(torchvision.utils.make_grid(images))
print(f'GroundTruth: {labels}')
```

```
Predicted: tensor([0, 0, 2, 0])
```



```
GroundTruth: tensor([0, 0, 2, 0])
```

Next steps

Furture evaluation could be done on this assignmnet. For example see the confusion matrix and see compare the accuracy of the model for each classes. The dataset is relatively small and unbalanced (#rain=213, #cloudy=300, #sunrise=70, #shine=253), it could be augmented by flipping images. I used the simples CNN model, which was enough for this task. However, more complex networks could be investigated for classification.

In [ ]: