Lab ML For Data Science: Part III

Getting Insights into Images and their Metadata

1.

The Dataset

Dataset Description:

We work with image data of leaves, which are of two types : “Healthy” and “Black Rot”

Sample image from class “Black Rot”

Sample image from class “Healthy”

2.

Pretrained Models for Image Recognition

Preprocessing and calculating the direction of difference of means

- 200 images for each class in the train data and
- 50 images for each class in the test data.
- Transform the data and resize it as required for modelling.
- The preprocessed image tensor is passed through the features part of the pretrained VGG-16 model.
- The shape of features: [1, 512, 7, 7].
- Flatten the features to calculate the mean of each class given by the formula:

$$\mu_1 = \frac{1}{|C_1|} \sum_{i \in C_1} \Phi(x_i)$$

$$\mu_2 = \frac{1}{|C_2|} \sum_{i \in C_2} \Phi(x_i)$$

- The size of flattened mean: 25088.

The direction w of difference of means is given by:

$$w = \frac{\mu_2 - \mu_1}{\|\mu_2 - \mu_1\|}$$

Then we calculate the discriminant function giving us the score for any instance

$$g(x) = w^\top \Phi(x)$$

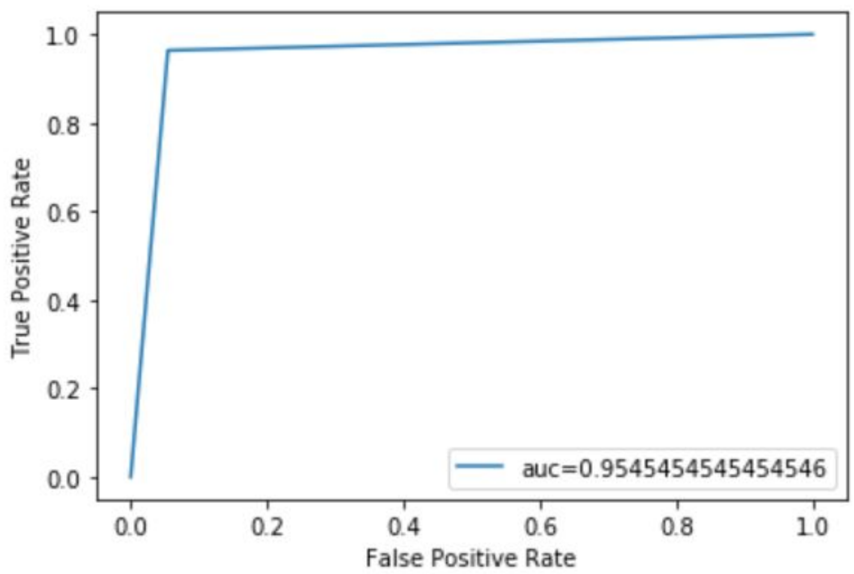
3.

Predicting Classes From Images

Computing the AUC score

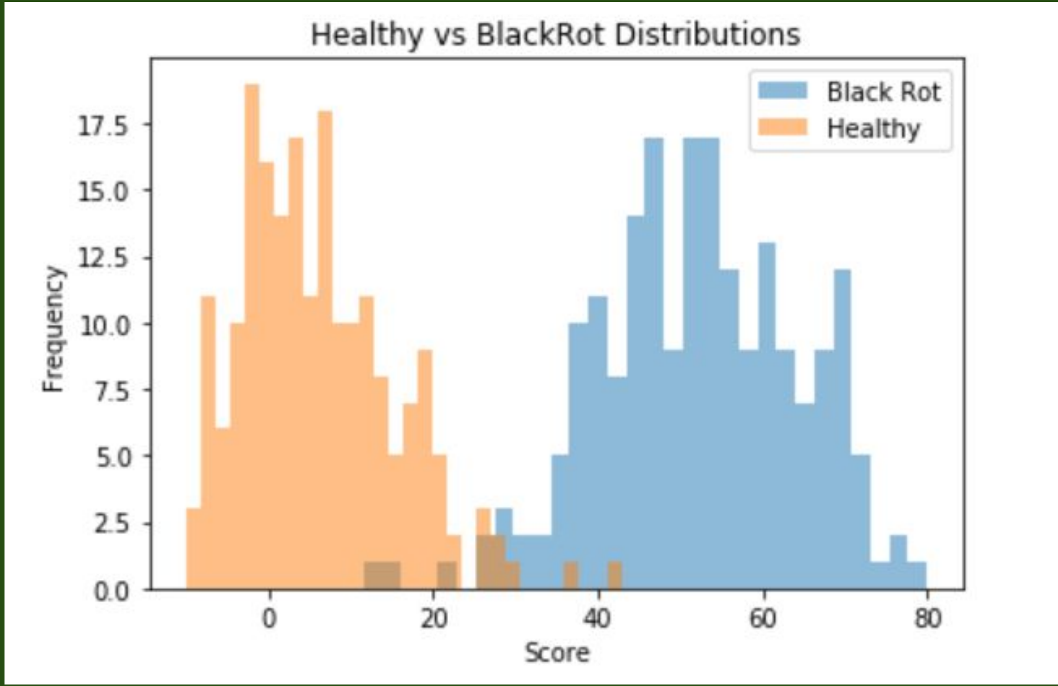
```
Threshold: 15.0
AUC Score: 0.8818181818181818
Threshold: 17.2222222222222
AUC Score: 0.9090909090909092
Threshold: 19.444444444444443
AUC Score: 0.9181818181818182
Threshold: 21.666666666666668
AUC Score: 0.9272727272727271
Threshold: 23.8888888888889
AUC Score: 0.9272727272727272
Threshold: 26.1111111111111
AUC Score: 0.9545454545454545
Threshold: 30.555555555555557
AUC Score: 0.9545454545454546
Best Threshold: 30.555555555555557
Best AUC Score: 0.9545454545454546
```

```
The number of instances used for class Healthy:  56
The number of instances used for class Black Rot:  54
```



We computed the optimum AUC score by iterating over a range of thresholds separating the 2 classes.
The AUC curve for the best threshold is as shown above.

Classification based on scores



We classify the instances by comparing their scores to the threshold value.

4.1.

Sensitivity Analysis

Getting Pixel Wise Contributions:

We now apply our model on a test image and understand which pixels contribute the most to the classification of the image.

```
# Preprocess the input image
preprocess = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
])

# Load and preprocess the image
image_path = 'data_TAD/test/Black_Rot/image (542).JPG'
image = Image.open(image_path).convert('RGB')
input_tensor = preprocess(image)
input_batch = input_tensor.unsqueeze(0)
print(input_batch.shape)
```

```
torch.Size([1, 3, 224, 224])
```

We first preprocess the image, so we have 224 * 224 pixels for each of the three channels (R, G, B)

We then apply the existing model to get an output.

Getting Pixel Wise Contributions:

```
# Calculate the gradients
output[0, predicted_idx].backward()

# Get the gradients of the input tensor
gradients = input_batch.grad[0]

# print("input_batch",input_batch)
print("input_batch shape",input_batch.shape)
# print("gradients",gradients)
print("gradients shape",gradients.shape)
```

```
input_batch shape torch.Size([1, 3, 224, 224])
gradients shape torch.Size([3, 224, 224])
```

```
importance_scores = torch.norm(gradients, p=2, dim=0,keepdim=False)**2

importance_scores.shape
```

```
torch.Size([224, 224])
```

```
np.round(importance_scores, decimals=4)
```

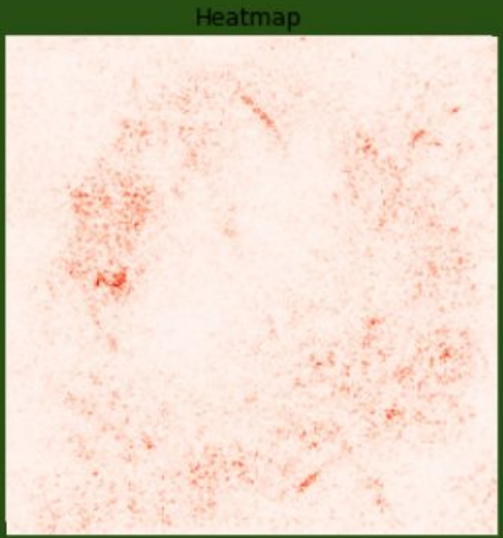
```
tensor([[1.0000e-04, 5.0000e-04, 7.0000e-04,  ..., 1.9000e-03, 3.0000e-04,
         1.0000e-04],
        [0.0000e+00, 5.0000e-04, 3.6000e-03,  ..., 5.2000e-03, 6.0000e-04,
         1.0000e-04],
        [1.0000e-04, 5.1000e-03, 1.4000e-03,  ..., 2.4000e-03, 3.0000e-04,
         1.0000e-04],
        ...,
        [0.0000e+00, 3.0000e-04, 4.1000e-03,  ..., 1.0000e-03, 4.0000e-04,
         1.0000e-04],
        [1.9000e-03, 2.1000e-03, 3.6000e-03,  ..., 8.0000e-04, 7.0000e-04,
         0.0000e+00],
        [3.0000e-04, 2.0000e-03, 1.0000e-03,  ..., 2.0000e-04, 1.0000e-04,
         4.0000e-04]])
```

We then calculate the gradient vector across the three channels (R,G,B).

Then we accumulate the gradient vector along each of our three directions (R,G,B) by calculating their square norm, so we can have a final importance score of size 224 * 224

These scores represent the importance of each pixel in the input image.

Getting Pixel Wise Contributions:



To visualize the importance scores, we normalize them and convert them to a heatmap image.

The heatmap image highlights important features in the original image.

However, sensitivity analysis using gradients tends to produce noisy and not fully satisfactory explanations.

4.2.

More Robust Explanations

Robustify gradient-based explanations

To robustify the features , we now focus on excitatory effects and less on inhibitory effects in the network.

$$z_k = \left(\sum_j a_j w_{jk}^{\uparrow} + b_k^{\uparrow}\right) \cdot \left[\frac{\sum_j a_j w_{jk} + b_k}{\sum_j a_j w_{jk}^{\uparrow} + b_k^{\uparrow}}\right]_{\text{cst.}}$$

$$w_{jk}^{\uparrow} = w_{jk} + 0.25 \max(0, w_{jk})$$
$$b_k^{\uparrow} = b_k + 0.25 \max(0, b_k).$$

- To achieve this, we rewrite specific layers in a way that the forward function remains the same locally but the gradient is modified to implement the asymmetry.
- By biasing the gradient in this way, the network tends to prioritize the learning of features and patterns that have a positive impact on the task at hand, while downplaying the influence of features that may hinder its performance.

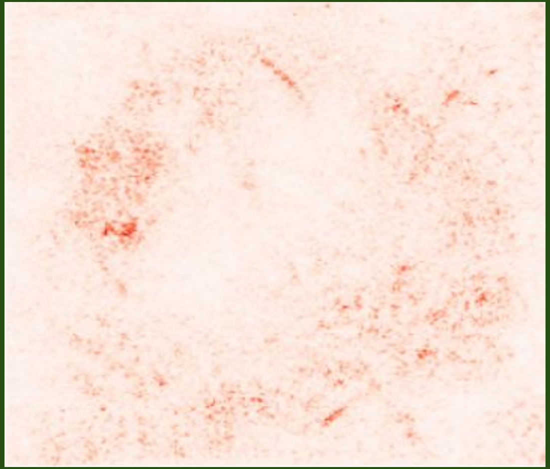
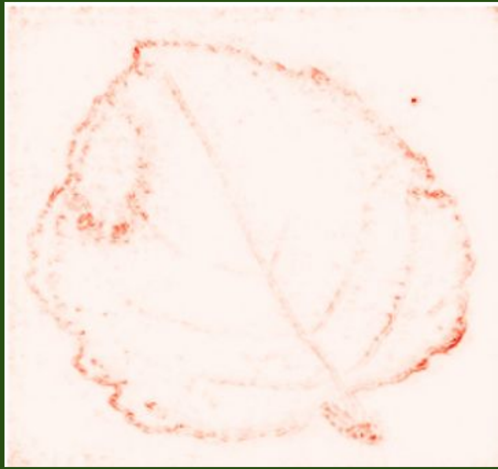
Excitatory effects over inhibitory effects

```python
class BiasedLayer(nn.Module):
    def __init__(self, original_layer,gamma:float):
        super(BiasedLayer, self).__init__()

        # Clone the original layer
        self.layer = original_layer
        self.gamma = gamma
        # self.biased_layer = self.clone_layer_with_bias(original_layer)
        self.biased_layer = deepcopy(self.layer)
        self.biased_layer.bias = nn.Parameter(self.layer.bias+ self.gamma * torch.relu(self.layer.bias))
        self.biased_layer.weight = nn.Parameter(self.layer.weight + self.gamma * torch.relu(self.layer.weight))


    def forward(self, x):

        original_output = self.layer(x)
        biased_output = self.biased_layer(x)
        biased_output_detached = biased_output.detach()
        original_output_detached = original_output.detach()

        scaling_factor = original_output_detached / biased_output_detached

        output = biased_output * scaling_factor

        return output

#
# Modify the VGG-16 model by replacing certain layers with biased versions
modified_model = model_vgg16_features
for name, module in modified_model.named_children():
    if isinstance(module, nn.Linear) or isinstance(module, nn.Conv2d):
        modified_model._modules[name] = BiasedLayer(module,gamma=0.25)
```

- As explained before, we modify the layers in a way that we get the gradient which favors the excitatory effects keeping the output same as the original VGG features model.

```
Sequential(
  (0): BiasedLayer(
    (layer): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (biased_layer): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  )
  (1): ReLU(inplace=True)
  (2): BiasedLayer(
    (layer): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (biased_layer): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  )
  (3): ReLU(inplace=True)
  (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (5): BiasedLayer(
    (layer): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

Feature comparisons

- The initial heatmap of the image is noisy while after modifying the gradient, we get a sharper image which accentuates the boundaries/features of the leaf. Although considering the task at hand, this does not distinguishes the part which has the disease.
- One idea to improve is to have an semantic segmentation model to identify individual objects within an image and assign a separate mask to each object.