In [1]:
```
import os
import glob
import numpy as np
import cv2
import pandas as pd
import matplotlib.pyplot as plt
from skimage.filters import threshold_otsu
import random
import csv
#import openpyxl


import tqdm
import pickle
from sklearn.metrics import classification_report, accuracy_score, confusio
n_matrix,log_loss
```

In [2]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision.models as models
import torchvision.datasets as datasets
import torchvision
from torchvision import transforms


import pytorch_lightning as pl
import albumentations as A
from albumentations.pytorch import ToTensorV2
import timm
from collections import OrderedDict


from torch.autograd import Variable
```

In [3]:
```
def show_thumbnail(image,thumbnail):
    f, axes = plt.subplots(1, 2, figsize=(20, 10));
    ax = axes.ravel();
    ax[0].imshow(image, cmap='gray');
    ax[0].set_title('Original');
    ax[1].imshow(thumbnail, cmap='gray')
    ax[1].set_title('thumbnail');
```

In [4]:
```
def show_thumbnail1(image,thumbnail,img3):
    f, axes = plt.subplots(1, 3, figsize=(20, 10));
    ax = axes.ravel();
    ax[0].imshow(image, cmap='gray');
    ax[0].set_title('Original');
    ax[1].imshow(thumbnail, cmap='gray')
    ax[1].set_title('pred');
    ax[2].imshow(img3, cmap='gray')
    ax[2].set_title('patch');
```

In [5]:
```
patch_size = 224
```

In [6]:
```
#WSI_Classification
```

In [7]:
```
class resnet50(nn.Module):
    def __init__(self,num_classes = None):
        super(resnet50,self).__init__()
        self.model = torch.hub.load('pytorch/vision:v0.10.0', 'resnet50', p
retrained=True)
        self.num_classes = num_classes


        self.model.fc = nn.Sequential(
              nn.Linear(in_features= self.model.fc.in_features, out_feature
s=1000, bias=True),
              nn.Dropout(0.4),
              nn.LeakyReLU(),
              nn.BatchNorm1d(1000, eps=1e-05, momentum=0.1, affine=True, tr
ack_running_stats=True),
              nn.Dropout(p=0.5),
              nn.Linear(in_features=1000, out_features=self.num_classes, bi
as=True),
              )

        #self.sigmoid = nn.Sigmoid()


    def forward(self, x):
        x = self.model(x).view(-1)
        return x
```

In [8]:
```
resnet50_model1 = torch.load('/home2/krishna.chandra/Scrap/colon_cancer/cla
ssification_results/model_weights/resnet50_WSI_Level_224.h5')
```

In [9]:
```
resnet50_model1 = resnet50_model1.eval()
img = torch.randn((1,3,patch_size,patch_size)).to('cuda',dtype= torch.float
32)
op = resnet50_model1(img)
print(op)
```

```
tensor([15.4982], device='cuda:0', grad_fn=<ViewBackward0>)
```

In [10]:
```
## CancerousWSI Classification
```

In [11]:
```
# ConvexNet = torch.load('/home/swathiguptha/colon_cancer_dataset/classific
ation_results/model_weights/convnext_base_50epochs_2classes_224_v01.h5')
# ConvexNet = ConvexNet.eval()
```

In [12]:
```
# resnet50_model2 = torch.load('/home/swathiguptha/colon_cancer_dataset/cla
ssification_results/model_weights/resnet50_50epochs_2classesCancWSI_resnet_
224_v01.h5')
# resnet50_model2.eval()
```

In [13]:
```
from transpath_classification import *
sg = nn.Sigmoid()
```

In [14]:
```
# transpath_model = torch.load('/home2/krishna.chandra/Scrap/colon_cancer/classification_results/model_weights/transpath_50epochs_2classes_224_v09.h5')
transpath_model = torch.load('/home2/krishna.chandra/Scrap/transpath_50epochs_2classes_224_v06.h5')
transpath_model = transpath_model.eval()
```

In [15]:
```
img = torch.randn((1,3,224,224)).to('cuda',dtype= torch.float32)
op = transpath_model(img)
print(sg(op))
```

```
tensor([0.6001], device='cuda:0', grad_fn=<SigmoidBackward0>)
```

In [16]:
```
#Segmentation
```

In [17]:
```
class Attention(nn.Module):
    def __init__(self,dim,n_heads=12,qkv_bias=False,attn_p=0.,proj_p = 0.):
        super(Attention,self).__init__()
        self.dim = dim                              #dimension of each patch
of image
        self.n_heads = n_heads                      #number of heads in MHA
        self.head_dim = dim//n_heads                #the lenght of q,k,v for
each patch

        self.scale = self.head_dim ** -0.5

        self.qkv = nn.Linear(dim,dim*3,bias = qkv_bias)
        self.attn_drop = nn.Dropout(attn_p)
        self.proj = nn.Linear(dim,dim)
        self.proj_drop = nn.Dropout(proj_p)

    def forward(self,x):
        n_samples,n_tokens,dim = x.shape

        qkv = self.qkv(x)
        qkv = qkv.reshape(n_samples,n_tokens,3,self.n_heads,self.head_dim)
        qkv = qkv.permute(2,0,3,1,4)

        q,k,v = qkv[0],qkv[1],qkv[2]

        k_t = k.transpose(-2,-1)

        dp = (q@k_t)*self.scale

        attn = dp.softmax(dim=-1)
        attn = self.attn_drop(attn)
        weighted_avg = attn@v
        weighted_avg = weighted_avg.transpose(1,2)
        weighted_avg = weighted_avg.flatten(2)
        x = self.proj(weighted_avg)
        x = self.proj_drop(x)

        return x

class MLP(pl.LightningModule):
    def __init__(self,in_features,hidden_features,out_features,p=0.):
        super(MLP,self).__init__()
        self.fc1 = nn.Linear(in_features,hidden_features)
        self.act = nn.GELU()
        self.fc2 = nn.Linear(hidden_features,out_features)
        self.drop = nn.Dropout(p)

    def forward(self,x):
        x = self.fc1(x)
        x = self.act(x)
        x = self.drop(x)
        x = self.fc2(x)
        x = self.drop(x)

        return x

class Block(nn.Module):
    def __init__(self,dim,n_heads,mlp_ratio=4,qkv_bias=True,p=0.,attn_p=
0.):
```

```
        super(Block,self).__init__()
        self.norm1 = nn.LayerNorm(dim,eps=1e-6)
        self.attn = Attention(dim=dim,n_heads = n_heads,qkv_bias = qkv_bia
s,attn_p = attn_p,proj_p = p)
        self.norm2 = nn.LayerNorm(dim,eps=1e-6)
        hidden_features = int(dim*mlp_ratio) #3072
        self.mlp = MLP(in_features = dim,hidden_features = hidden_features,
out_features=dim)

    def forward(self,x):

        x = x + self.attn(self.norm1(x))
        x = x + self.mlp(self.norm2(x))

        return x

def conv_trans_block(in_channels,out_channels):
    conv_trans_block = nn.Sequential(
    nn.Conv2d(in_channels,out_channels,kernel_size=3,padding=1),
    nn.ReLU(inplace=True),
    nn.BatchNorm2d(out_channels)
    )
    return conv_trans_block

from collections import OrderedDict

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



def np2th(weights, conv=False):
    """Possibly convert HWIO to OIHW."""
    if conv:
        weights = weights.transpose([3, 2, 0, 1])
    return torch.from_numpy(weights)


class StdConv2d(nn.Conv2d):

    def forward(self, x):
        w = self.weight
        v, m = torch.var_mean(w, dim=[1, 2, 3], keepdim=True, unbiased=Fals
e)
        w = (w - m) / torch.sqrt(v + 1e-5)
        return F.conv2d(x, w, self.bias, self.stride, self.padding,
                        self.dilation, self.groups)


def conv3x3(cin, cout, stride=1, groups=1, bias=False):
    return StdConv2d(cin, cout, kernel_size=3, stride=stride,
                     padding=1, bias=bias, groups=groups)


def conv1x1(cin, cout, stride=1, bias=False):
    return StdConv2d(cin, cout, kernel_size=1, stride=stride,
                     padding=0, bias=bias)
```

```
class PreActBottleneck(nn.Module):
    """Pre-activation (v2) bottleneck block.
    """

    def __init__(self, cin, cout=None, cmid=None, stride=1):
        super().__init__()
        cout = cout or cin
        cmid = cmid or cout//4

        self.gn1 = nn.GroupNorm(32, cmid, eps=1e-6)
        self.conv1 = conv1x1(cin, cmid, bias=False)
        self.gn2 = nn.GroupNorm(32, cmid, eps=1e-6)
        self.conv2 = conv3x3(cmid, cmid, stride, bias=False)  # Original co
de has it on conv1!!
        self.gn3 = nn.GroupNorm(32, cout, eps=1e-6)
        self.conv3 = conv1x1(cmid, cout, bias=False)
        self.relu = nn.ReLU(inplace=True)

        if (stride != 1 or cin != cout):
            # Projection also with pre-activation according to paper.
            self.downsample = conv1x1(cin, cout, stride, bias=False)
            self.gn_proj = nn.GroupNorm(cout, cout)

    def forward(self, x):

        # Residual branch
        residual = x
        if hasattr(self, 'downsample'):
            residual = self.downsample(x)
            residual = self.gn_proj(residual)

        # Unit's branch
        y = self.relu(self.gn1(self.conv1(x)))
        y = self.relu(self.gn2(self.conv2(y)))
        y = self.gn3(self.conv3(y))

        y = self.relu(residual + y)
        return y



class ResNetV2(nn.Module):
    """Implementation of Pre-activation (v2) ResNet mode."""

    def __init__(self, block_units, width_factor):
        super().__init__()
        width = int(64 * width_factor)
        self.width = width

        self.root = nn.Sequential(OrderedDict([
            ('conv', StdConv2d(3, width, kernel_size=7, stride=2, bias=Fals
e, padding=3)),
            ('gn', nn.GroupNorm(32, width, eps=1e-6)),
            ('relu', nn.ReLU(inplace=True)),
            # ('pool', nn.MaxPool2d(kernel_size=3, stride=2, padding=0))
        ]))
```

```
        self.body = nn.Sequential(OrderedDict([
            ('block1', nn.Sequential(OrderedDict(
                [('unit1', PreActBottleneck(cin=width, cout=width*4, cmid=w
idth))] +
                [(f'unit{i:d}', PreActBottleneck(cin=width*4, cout=width*4,
cmid=width)) for i in range(2, block_units[0] + 1)],
                ))),
            ('block2', nn.Sequential(OrderedDict(
                [('unit1', PreActBottleneck(cin=width*4, cout=width*8, cmid
=width*2, stride=2))] +
                [(f'unit{i:d}', PreActBottleneck(cin=width*8, cout=width*8,
cmid=width*2)) for i in range(2, block_units[1] + 1)],
                ))),
            ('block3', nn.Sequential(OrderedDict(
                [('unit1', PreActBottleneck(cin=width*8, cout=width*16, cmi
d=width*4, stride=2))] +
                [(f'unit{i:d}', PreActBottleneck(cin=width*16, cout=width*1
6, cmid=width*4)) for i in range(2, block_units[2] + 1)],
                ))),
        ]))

    def forward(self, x):
        features = []
        b, c, in_size, _ = x.size()
        x = self.root(x)
        features.append(x)
        x = nn.MaxPool2d(kernel_size=3, stride=2, padding=0)(x)
        for i in range(len(self.body)-1):
            x = self.body[i](x)
            right_size = int(in_size / 4 / (i+1))
            if x.size()[2] != right_size:
                pad = right_size - x.size()[2]
                assert pad < 3 and pad > 0, "x {} should {}".format(x.size
(), right_size)
                feat = torch.zeros((b, x.size()[1], right_size, right_siz
e), device=x.device)
                feat[:, :, 0:x.size()[2], 0:x.size()[3]] = x[:]
            else:
                feat = x
            features.append(feat)
        x = self.body[-1](x)
        return x, features[::-1]

class Embeddings(nn.Module):
    """Construct the embeddings from patch, position embeddings.
    """
    def __init__(self,embed_dim = 768,n_patches=196,p=0.,in_channels=3):
        super(Embeddings, self).__init__()
        self.hybrid_model = ResNetV2(block_units=(3, 4, 9), width_factor=1)
        in_channels = self.hybrid_model.width * 16
        self.patch_embeddings = nn.Conv2d(in_channels=in_channels,
                                          out_channels=embed_dim,
                                          kernel_size=1,
                                          stride=1)
        self.position_embeddings = nn.Parameter(torch.zeros(1, n_patches, e
mbed_dim))
        self.dropout = nn.Dropout(p=p)
```

```
    def forward(self, x):
        x, features = self.hybrid_model(x)
        x = self.patch_embeddings(x)  # (B, hidden. n_patches^(1/2), n_patc
hes^(1/2))
        x = x.flatten(2)
        x = x.transpose(-1, -2)  # (B, n_patches, hidden)
        embeddings = x + self.position_embeddings
        embeddings = self.dropout(embeddings)
        return embeddings, features


class transUnet(pl.LightningModule):
    def __init__(self,img_size = 224,patch_size=16,in_channels = 3,n_classe
s=1,embed_dim = 768,depth=12,n_heads=12,mlp_ratio=4.,qkv_bias=True,p=0.,att
n_p=0.):
        super(transUnet,self).__init__()

        self.embed_dim = embed_dim
        self.img_size = img_size



        self.n_patches = (img_size//patch_size) ** 2
        self.embeddings = Embeddings(embed_dim,self.n_patches,p)


        self.blocks = nn.ModuleList(
                        [
                            Block(dim = embed_dim,
                                  n_heads=n_heads,
                                 mlp_ratio=mlp_ratio,
                                 qkv_bias = qkv_bias,
                                 p=p,
                                 attn_p=attn_p,)

                            for _ in range(depth)
                        ])


        #decoder part
        self.deconv1 = conv_trans_block(embed_dim,512)

        self.deconv2_1 = conv_trans_block(1024,256)
        self.deconv2_2 = conv_trans_block(256,256)

        self.deconv3_1 = conv_trans_block(512,128)
        self.deconv3_2 = conv_trans_block(128,128)


        self.deconv4_1 = conv_trans_block(192,64)
        self.deconv4_2 = conv_trans_block(64,64)

        self.upsample = nn.UpsamplingBilinear2d(scale_factor=2)

        self.prefinal_1 = conv_trans_block(64,16)
        self.prefinal_2 = conv_trans_block(16,16)

        self.out = nn.Conv2d(16,1,kernel_size=1)
        self.sigmoid = nn.Sigmoid()
```

```
        self.losses = []
    def forward(self,x):

        n_samples = x.shape[0]


        x = self.embeddings(x)


        projections = x[0]
        features = x[1]

        for block in self.blocks:
            projections = block(projections)


        projections = projections.transpose(1,2)
        projections = projections.reshape(n_samples,self.embed_dim,int(sel
f.img_size/16),int(self.img_size/16))

        x1 = projections

        x1 = self.deconv1(x1)     #(n,512,224/16,224/16)
        x1 = self.upsample(x1)    #(n,512,224/8,224/8)
        x1 = self.deconv2_1(torch.cat([x1,features[0]],1))
        x1 = self.deconv2_2(x1)


        x1 = self.upsample(x1)    #(n,256,224/4,224/4)

        x1 = self.deconv3_1(torch.cat([x1,features[1]],1))
        x1 = self.deconv3_2(x1)

        x1 = self.upsample(x1)    #(n,128,224/2,224/2)

        x1 = self.deconv4_1(torch.cat([x1,features[2]],1))
        x1 = self.deconv4_2(x1)

        x1 = self.upsample(x1)   #(n,64,224,224)
        x1 = self.prefinal_1(x1)
        x1 = self.prefinal_2(x1)

        x1 = self.out(x1)
        x1 = self.sigmoid(x1)
        return x1
```

In [18]:
```
m1 = timm.create_model('vit_base_patch16_384',pretrained='True')
transunet_model = transUnet(p=0.4,attn_p=0.4)    #512
```

In [19]:
```
transunet_model = transunet_model.load_from_checkpoint('/home2/krishna.chan
dra/Scrap/colon_cancer/segmentation_results/weights/vit_pt-epoch=28-cancwsi
-patches-224=0.5905.ckpt')
#transunet_model = transunet_model.eval()
```

In [20]:
```
transunet_model = transunet_model.to('cuda').eval()
```

In [21]:
```
img = torch.randn((1,3,224,224)).to('cuda',dtype= torch.float32)
op = transunet_model(img)
print(op.shape)
op = op.detach().cpu().numpy().reshape(op.shape[2],op.shape[3],op.shape[0])
print(op.shape)
#show_thumbnail(op,op)
```

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

In [22]:
```
positive_test_patients = ['D20180498101_2019-05-14 14_58_30', 'D20180715202
_2019-05-21 13_14_56', 'D20180792401_2019-05-21 13_38_34','D20181010201_201
9-05-21 14_08_09','F2018_13446_1-1_2019-02-20 21_33_15']
```

In [23]:
```
negative_test_patients = ['D20181046602_2019-06-10 11_41_40', 'D20181047302
_2019-06-10 11_38_18', 'D20181168805_2019-06-10 11_18_54', 'D20181187110_20
19-06-10 11_15_43', 'D20181284101_2019-06-10 10_53_49', 'D20190284705_2019-
06-10 15_10_34', 'D20190284802_2019-06-10 15_06_21', 'D20190284902_2019-06-
10 15_02_38', 'D20190286206_2019-06-10 14_58_35', 'D20190296201_2019-06-10
14_50_28', 'D20190381406_2019-06-10 14_42_20', 'D20190399001_2019-06-10 14_
39_51', 'D20190399101_2019-06-10 14_34_19', 'D20190445003_2019-06-10 14_30_
16', 'D20190445103_2019-06-10 14_26_02']

print(len(negative_test_patients))
```

```
15
```

In [24]:
```
canc_wsi_dir = '/scratch/normalised_wsi/tissue-train-pos-v1'
non_canc_wsi_dir = '/scratch/normalised_wsi/tissue-train-neg'
```

In [25]:
```
pos_wsis = glob.glob(canc_wsi_dir+'/*.jpg')
neg_wsis = glob.glob(non_canc_wsi_dir+'/*.jpg')
print(len(pos_wsis),len(neg_wsis))
```

```
500 410
```

In [26]:
```
pos_test_WSIs = [ele for ele in pos_wsis if ele.split('/')[-1].split('-lv1-
')[0] in positive_test_patients]
neg_test_WSIs = [ele for ele in neg_wsis if ele.split('/')[-1].split('-lv1-
')[0] in negative_test_patients]

print(len(pos_test_WSIs),len(neg_test_WSIs))
```

```
12 29
```

In [27]:
```
patch_size = 224
overlap = 0
stride = 224
```

In [28]:
```
valid_transform=transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])
```

In [29]:
```
def generate_mask(wsi,mask,gt,patch_positions,patch_size,model):

    val_transform=A.Compose([ToTensorV2(),])
    print(len(patch_positions))
    for i in (range(len(patch_positions))):
        row = patch_positions[i][0]
        col = patch_positions[i][1]
        patch = wsi[row:row+patch_size,col:col+patch_size,:]
        gt_patch = gt[row:row+patch_size,col:col+patch_size]
        transformed = val_transform(image = patch)
        transformed_patch = (transformed['image']/255.).float().unsqueeze
(0).to('cuda')
        #print('inp:',transformed_patch.shape)
        pred_mask = model(transformed_patch)
        pred_mask = pred_mask.detach().cpu().numpy().reshape(pred_mask.shap
e[2],pred_mask.shape[3],pred_mask.shape[0])*255
        pred_mask = pred_mask.astype('uint8')
        ret2, pred_thresh = cv2.threshold(pred_mask, 52, 255, cv2.THRESH_BI
NARY) #thresholding to get a binary mask
        #print('datatype:',pred_mask.dtype,pred_thresh.dtype)
        #print('op shape:',pred_mask[:2,:2,:],'thresh:',pred_thresh[:2,:2])
        mask[row:row+patch_size,col:col+patch_size] = np.logical_or(mask[ro
w:row+patch_size,col:col+patch_size], pred_thresh)
        #show_thumbnail(gt,mask)
        #show_thumbnail1(gt_patch,pred_thresh,patch)

    return mask
```

In [30]:
```
def make_kernel_dataset(patch_positions,WSI,gt,tarnspath_model):

    patch_size = 224
    kernel = np.ones((5,5), np.uint8)
    val_transforms=transforms.Compose([ transforms.ToTensor()])


    gray_wsi = cv2.cvtColor(WSI,cv2.COLOR_RGB2GRAY)
    ret1, wsi_thresh = cv2.threshold(gray_wsi, 0, 255, cv2.THRESH_BINARY_IN
V + cv2.THRESH_OTSU)

    x_data = []
    y_data = []
    final_positions = []




    x_data2 = []
    y_data2 = []
    final_positions2 = []

    for idx in range(len(patch_positions)):

        i = patch_positions[idx][0]
        j = patch_positions[idx][1]



        patch_flag = 0
        discarded_patches = 0
        non_tissue_count = 0
        temp = []

        for x in range(-1,2):
            for y in range(-1,2):

                #Discarding the edge cases
                p1 = i + patch_size*x if i + patch_size*x >= 0 and i + patc
h_size*x <= width - patch_size else -1
                p2 = j + patch_size*y if j + patch_size*y >= 0 and j + patc
h_size*y <= height - patch_size else -1



                if p1 == -1 or p2 == -1:
                    patch_flag = 1
                    break



                #Discarding the non_tissue
                tissue = wsi_thresh[p1:p1+patch_size,p2:p2+patch_size]


                if np.mean(tissue) < 60: # selecting only those patches whi
ch have atleast 60% tissue area
                    non_tissue_count += 1
```

```
                if non_tissue_count >= 2:
                    #print('Discarded:',total_patches)
                    patch_flag = 1
                    break

                patch = WSI[p1:p1+patch_size,p2:p2+patch_size,:]
                transformed_img = val_transforms(patch)
                patch_tensor = transformed_img.unsqueeze(0).to('cuda',dtype
= torch.float32)
                op = tarnspath_model(patch_tensor)
#                 op = sg(op)
                temp.append(op.detach().item())

            if patch_flag == 1:
                discarded_patches+=1
                break


        mask = gt[i:i+patch_size,j:j+patch_size]
        # To smoothen the edges of the cancerous
        img_erosion = cv2.erode(mask, kernel, iterations=5)
        img_dilation = cv2.dilate(img_erosion, kernel, iterations=5)
        y_temp = None
        if np.mean(img_dilation)>20: #Takes only those patches which contai
ns atleast 20% of cancerous regions
            y_temp = 0
        else:
            y_temp = 1



        if patch_flag==0:
            x_data.append(np.array(temp))
            y_data.append(y_temp)
            final_positions.append([i,j])

        else:
            patch = WSI[i:i+patch_size,j:j+patch_size,:]
            transformed_img = val_transforms(patch)
            patch_tensor = transformed_img.unsqueeze(0).to('cuda',dtype= to
rch.float32)
            op = tarnspath_model(patch_tensor)
            op = sg(op)
            if (op.detach().item())>=0.8:
                x_data2.append(1)
                y_data2.append(y_temp)
            else:
                x_data2.append(0)
                y_data2.append(y_temp)
                final_positions2.append([i,j])

    x_data = np.array(x_data)
    y_data = np.array(y_data)

    x_data2 = np.array(x_data2)
    y_data2 = np.array(y_data2)

    return final_positions,x_data,y_data,final_positions2,x_data2,y_data2
```

In [31]:
```
def eval(x_data,y_data,kernel,positions):

    # load the model from disk
    loaded_model = pickle.load(open('/home2/krishna.chandra/Scrap/colon_cancer/kernel_classification_data/finalized_model1.sav', 'rb'))

    data_predict = loaded_model.predict_proba(x_data) #Gives probablity for each class

#     preds= []

    final_positions = []

    for i in range(len(positions)):

        if data_predict[i][0] < 0.7:
            final_positions.append([positions[i][0],positions[i][1]])
#             preds.append(0)
#         else:
#             preds.append(1)

    print('final:',len(final_positions))

#     report = classification_report(y_data,preds)

#     print('-----------------classification-report-------------')
#     print(report)

#     import pandas as pd
#     confusion_matrix_df = pd.DataFrame(confusion_matrix(y_data,preds))

#     print(confusion_matrix_df)

    return final_positions
```

In [32]:
```
# def get_positions(non_canc_pos,canc_pos,model,WSI,gt):


#     patch_positions = canc_pos + non_canc_pos

#     print('initial:',len(patch_positions))
#     final_positions, x_data,y_data,final_positions2, x_data2,y_data2 = ma
ke_kernel_dataset(patch_positions,WSI,gt,model)

#     print('final for logistic:',len(final_positions),x_data.shape)
#     print('final not for logistic:',len(final_positions2),x_data2.shape)

#     final_positions = eval(x_data,y_data,1,final_positions)



#     print(len(final_positions))

#     print(len(final_positions+final_positions2))


#     return final_positions+final_positions2
```

In [33]:
```
def get_positions(non_canc_pos,canc_pos,model,WSI,gt):


    patch_positions = canc_pos + non_canc_pos
    final_positions = []
    #print('non_canc:',len(non_canc_pos),' canc:',len(canc_pos),'total: ',l
en(patch_positions)), transforms.Normalize([0.485, 0.456, 0.406], [0.229,
0.224, 0.225])
    valid_transform1=transforms.Compose([transforms.ToTensor() ])

    print(len(patch_positions))
    for i in range(len(patch_positions)):

        row = patch_positions[i][0]
        col = patch_positions[i][1]
        patch = WSI[row:row+patch_size,col:col+patch_size,:]
        gt_patch = gt[row:row+patch_size,col:col+patch_size]

#         transformed_img = valid_transform1((torchvision.transforms.functi
onal.to_pil_image(patch)))
#         patch_tensor = transformed_img.unsqueeze(0).to('cuda',dtype= torc
h.float32)
#         op = model(patch_tensor)
#         op = sg(op).cpu().detach().item()


#         if op<0.5: #Not sure about this
#             final_positions.append([row,col])

        #print('canc region:',np.mean(gt_patch))

        if np.mean(gt_patch)>=20:
                final_positions.append([row,col])




    print('Total canc patches on eliminating FN are:',len(final_positions))
    return final_positions
```

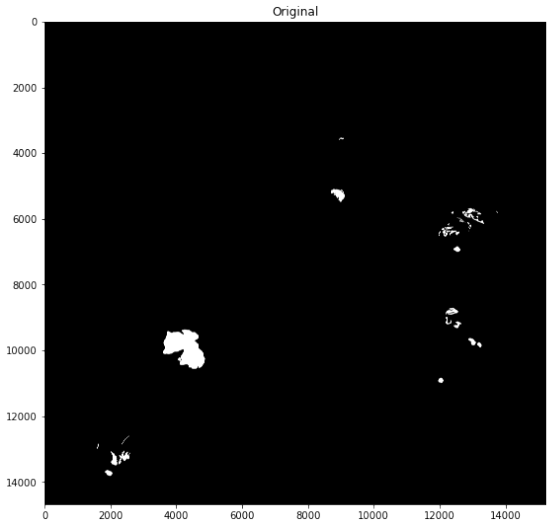
In [34]:
```
# def get_positions(non_canc_pos,canc_pos,model,WSI,gt):


#     patch_positions = canc_pos + non_canc_pos
#     final_positions = []
#     #print('non_canc:',len(non_canc_pos),' canc:',len(canc_pos),'total:
',len(patch_positions))
#     valid_transform=transforms.Compose([transforms.ToTensor())


#     for i in range(len(patch_positions)):

#         row = patch_positions[i][0]
#         col = patch_positions[i][1]
#         patch = WSI[row:row+patch_size,col:col+patch_size,:]
#         gt_patch = gt[row:row+patch_size,col:col+patch_size]

#         transformed_img = valid_transform((torchvision.transforms.functio
nal.to_pil_image(patch)))
#         patch_tensor = transformed_img.unsqueeze(0).to('cuda',dtype= torc
h.float32)
#         op = model(patch_tensor)
#         op = op.cpu().detach().item()


#         if op<0.5: #Not sure about this
#             final_positions.append([row,col])

#         #print('canc region:',np.mean(gt_patch))

# #         if np.mean(gt_patch)>20:
# #                 final_positions.append([row,col])


#     print('Total canc patches on eliminating FN are:',len(final_position
s))

#     return final_positions
```

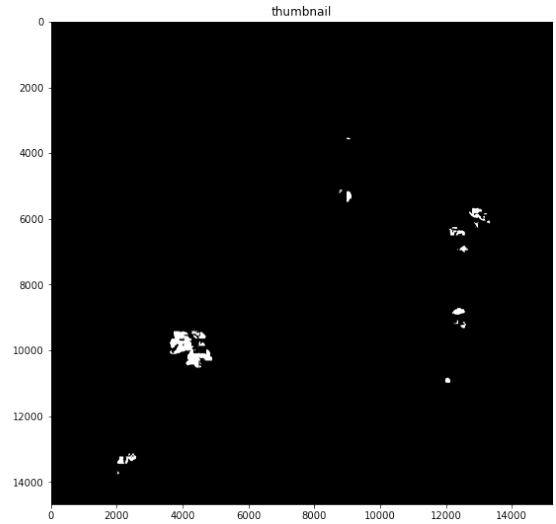
In [35]:
```
def dice_metric(inputs, target):

    target = target.reshape(-1)
    inputs = inputs.reshape(-1)
    intersection = (target * inputs)
    #print('intersection:',intersection,intersection.sum())
    dice = (2. * intersection.sum() ) / (target.sum() + inputs.sum() + 1e-
8)

    return dice
```

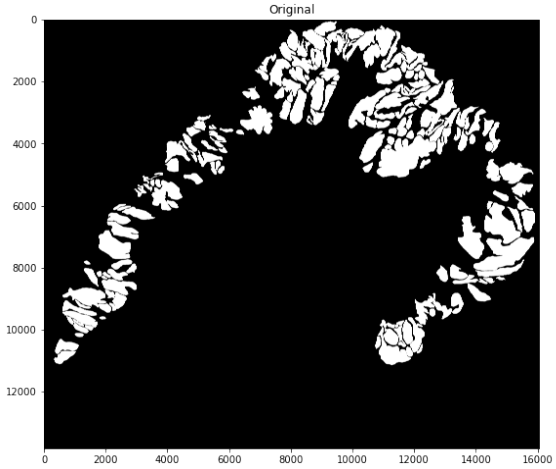
In [36]:
```
dice_score_sum = 0
canc_count = 0
non_canc_count = 0
score_sum = 0
for wsi in tqdm.tqdm(pos_test_WSIs):

    if not wsi.endswith("_mask.jpg"):

        #Creating a directory of cancerous and non-cancerous patches for ea
ch WSI
        classification = []
        classification_score = []

        #wsi = '/scratch/normalised_wsi/tissue-train-pos-v1/18-09530A_2019-
05-07 23_50_03-lv1-34626-18358-3736-6181.jpg'

        cancerous_mask = True
        wsi_name = wsi.split("/")[-1].split(".")[0]

        model_predictions = {0: [], 1: [],2:[]}

        #WSI
        wsi_img = cv2.imread(wsi)
        gray_wsi = cv2.cvtColor(wsi_img,cv2.COLOR_RGB2GRAY)
        ret1, wsi_thresh = cv2.threshold(gray_wsi, 0, 255, cv2.THRESH_BINAR
Y_INV + cv2.THRESH_OTSU)
        #show_thumbnail(gray_wsi,wsi_thresh)

        ground_truth = cv2.cvtColor(cv2.imread (wsi.replace('.jpg','_mask.j
pg')),cv2.COLOR_RGB2GRAY)
        ret2, ground_truth_thresh = cv2.threshold(ground_truth, 0, 255, cv
2.THRESH_BINARY ) #thresholding to get a binary mask
        #show_thumbnail(wsi_img,ground_truth_thresh)
        #print('datatype:',ground_truth.dtype, ground_truth_thresh.dtype)
        ##print('values:', ground_truth[:2,:2], ground_truth_thresh[:2,:2])

        width,height = wsi_img.shape[0],wsi_img.shape[1]

        for i in range(0,width,stride-overlap):
            for j in range(0, height, stride-overlap):

                i = width - patch_size if i > width or i + patch_size > wid
th else i
                j = height - patch_size if j > height or j+ patch_size > he
ight else j


                tissue = wsi_thresh[i:i+patch_size,j:j+patch_size]

                if np.mean(tissue)>= 20: # selecting only those patches whi
ch have atleast 60% tissue area

                    patch = wsi_img[i:i+patch_size,j:j+patch_size,:]
                    transformed_img = valid_transform((torchvision.transfor
ms.functional.to_pil_image(patch)))
                    patch_tensor = transformed_img.unsqueeze(0).to('cuda',d
type= torch.float32)
                    op = resnet50_model1(patch_tensor)
                    op = sg(op).cpu().detach().numpy()
```

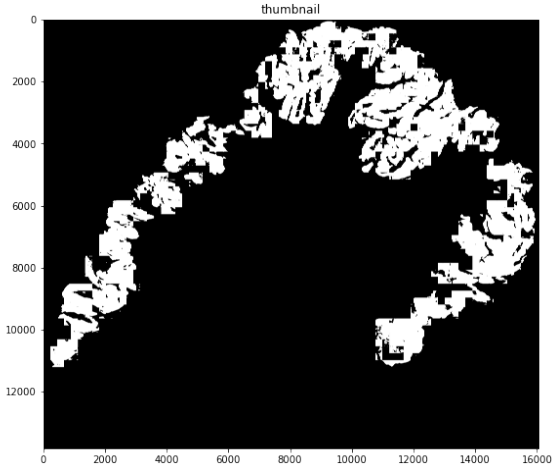
```
                    if op<0.5:
                        model_predictions[0].append(op) #non Cancerous
                        model_predictions[2].append([i,j])
                    else:
                        model_predictions[1].append([i,j]) #Cancerous



        total_patches = len(model_predictions[0])+len(model_predictions[1])
        total_nc_patches = len(model_predictions[0])
        if total_nc_patches >= int(0.8*total_patches):
            score = np.array(model_predictions[0]).sum()
            score_sum = score_sum + (score/total_nc_patches)
            non_canc_count = non_canc_count + 1
            print("Non-cancerous WSI with a score of:",(score/total_nc_patc
hes)*100)
        else:
            print('Cancerous WSI')
            print('Total patches:', total_patches,'Non-Cancerous patches:',
len(model_predictions[0]),' Cancerous patches:',len(model_predictions[1]))
            original_mask = cv2.cvtColor(cv2.imread (wsi.replace('.jpg','_m
ask.jpg')),cv2.COLOR_RGB2GRAY)

            predicted_mask = np.zeros_like(original_mask)
            positions = get_positions(model_predictions[1],model_prediction
s[2],transpath_model,wsi_img,ground_truth_thresh)
            #print(len(positions))
            predicted_mask = generate_mask(wsi_img,predicted_mask,original_
mask,positions,patch_size,transunet_model)

            score = dice_metric(torch.from_numpy(predicted_mask), torch.fro
m_numpy(original_mask/255))
            dice_score_sum += score
            canc_count = canc_count +1
            show_thumbnail(original_mask,predicted_mask)
            print('Dice Score:',score)


if canc_count!=0 :
    print('Avg Dice Score:', dice_score_sum/canc_count)
elif non_canc_count != 0:
    print('Avg classification score:',score_sum/non_canc_count)
```

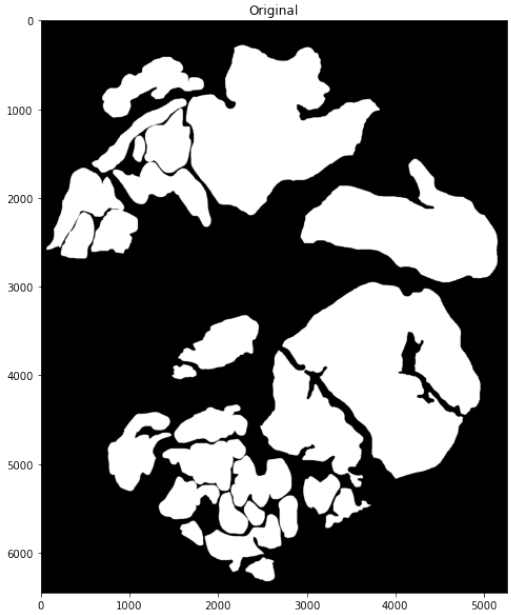
```
  0%|          | 0/12 [00:00<?, ?it/s]

Cancerous WSI
Total patches: 1869 Non-Cancerous patches: 43  Cancerous patches: 1826
1869
Total canc patches on eliminating FN are: 69
69

  8%|▊         | 1/12 [00:58<10:40, 58.27s/it]

Dice Score: tensor(0.7367, dtype=torch.float64)
Cancerous WSI
Total patches: 2953 Non-Cancerous patches: 1053  Cancerous patches: 1900
2953
Total canc patches on eliminating FN are: 927
927

 17%|█▋        | 2/12 [02:32<13:12, 79.28s/it]

Dice Score: tensor(0.8168, dtype=torch.float64)
Cancerous WSI
Total patches: 381 Non-Cancerous patches: 0  Cancerous patches: 381
381
Total canc patches on eliminating FN are: 326
326

 25%|██▌       | 3/12 [03:01<08:29, 56.58s/it]

Dice Score: tensor(0.9332, dtype=torch.float64)
Cancerous WSI
Total patches: 1460 Non-Cancerous patches: 5  Cancerous patches: 1455
1460
Total canc patches on eliminating FN are: 769
769

 42%|████▏     | 5/12 [04:28<05:43, 49.13s/it]

Dice Score: tensor(0.7972, dtype=torch.float64)
Cancerous WSI
Total patches: 528 Non-Cancerous patches: 1  Cancerous patches: 527
528
Total canc patches on eliminating FN are: 388
388

 58%|█████▊    | 7/12 [04:52<02:42, 32.54s/it]

Dice Score: tensor(0.9013, dtype=torch.float64)
Cancerous WSI
Total patches: 137 Non-Cancerous patches: 0  Cancerous patches: 137
137
Total canc patches on eliminating FN are: 123
123

100%|██████████| 12/12 [05:00<00:00, 25.05s/it]

Dice Score: tensor(0.8560, dtype=torch.float64)
Avg Dice Score: tensor(0.8402, dtype=torch.float64)
```

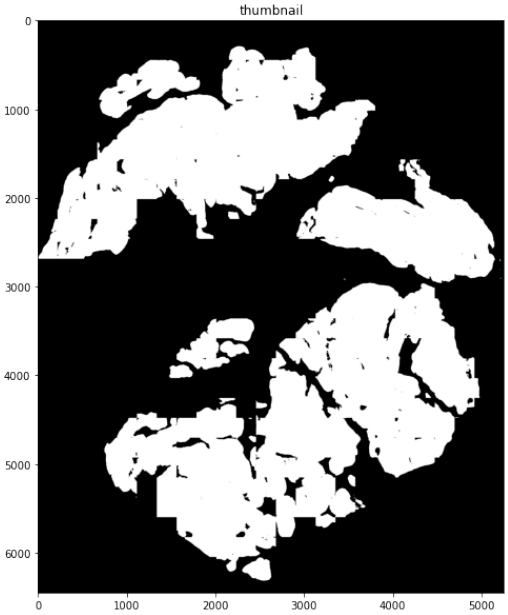

Original

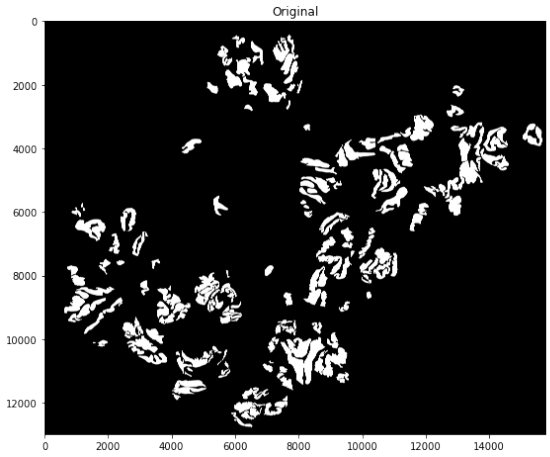

thumbnail

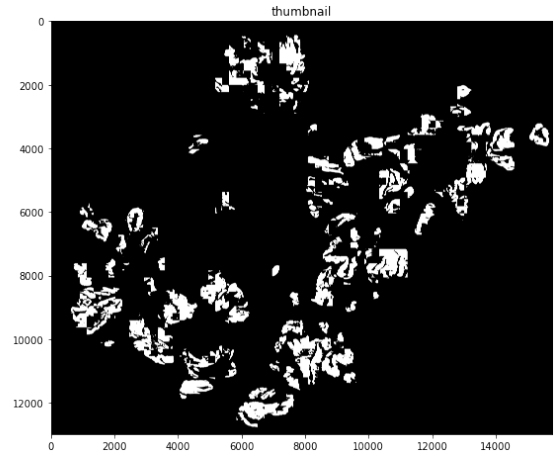

Original

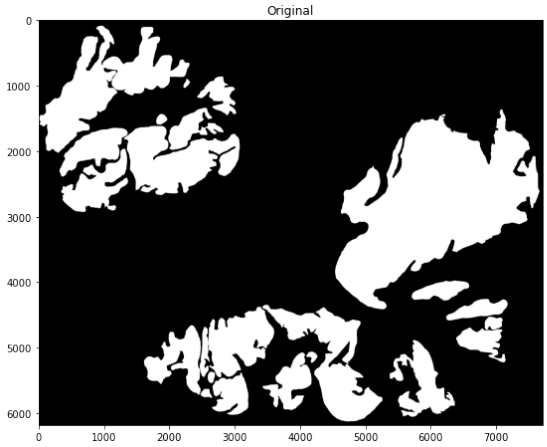

thumbnail

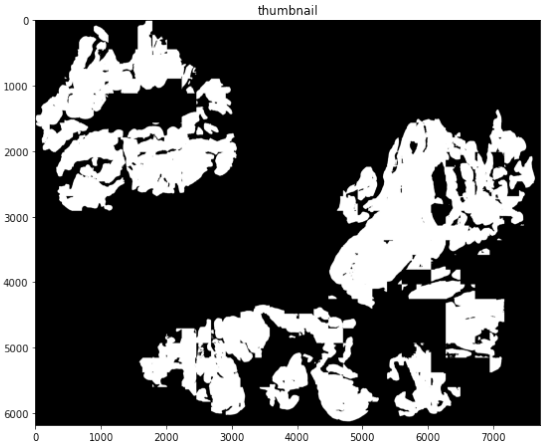

Original

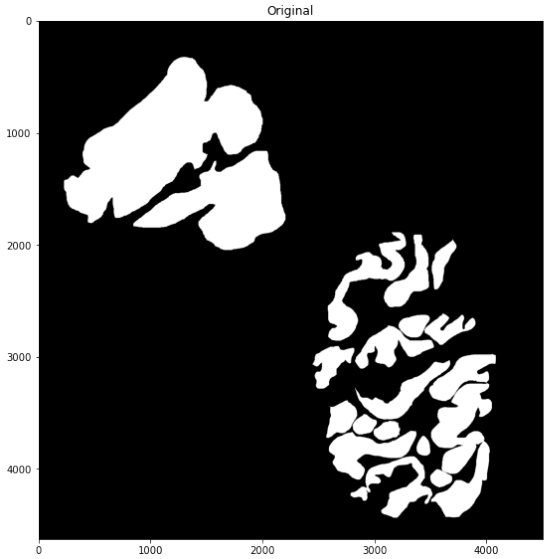

thumbnail

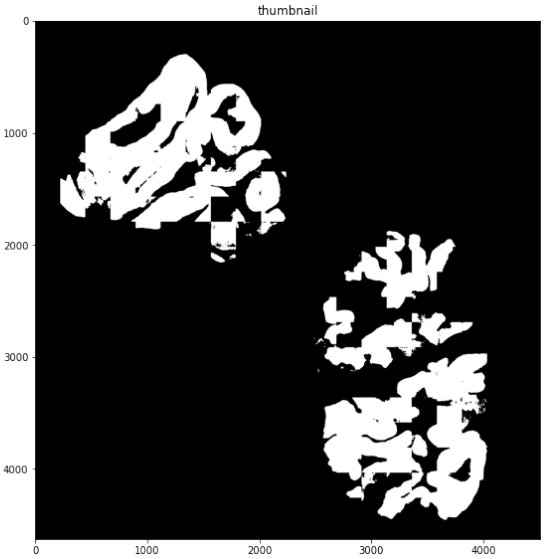

Original


thumbnail


Original


thumbnail


Original


thumbnail

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: