Playstores Project

First challenge was completed by using the .value_counts() and a few other pandas functions in python

Second challenge was completed also by using pandas functions in python and the suspicious companies selected were all the companies that were in unfriendly countries and had been removed from the play store by google

The review data was preprocessed, all nan values were removed and a perceptron was used to predict review rating based on the length of the review, length of the app name, objectivity, specificity, and sentiment. An alternate model was built that used purely the text in the review that took the text, tokenized it, converted them into a word embeddings matrix and fed into a neural network to predict rating. Both models overfit due to a heave skew in the data towards 4s.

Malware Analysis

The dataset of malware files was located and downloaded.

The benign files were separated from the malicious files and the executable permissions were disabled on the folder continaing malicious files. From there a script was created to preprocess both folders exe files into images using the hindex2xy hilbert space filling curve algorithm.

```python
def helper(hilb_idx: int, curr_order: int, hcurve_order: int, x: int, y: int) -> tuple:
    n1_hash = {0: (0, 0), 1: (0, 1), 2: (1, 1), 3: (1, 0)}

    # print(curr_order, x, y)

    if curr_order > hcurve_order:
        return int(x), int(y)

    l2_bits = hilb_idx & 3
    mult_constant = curr_order / 2

    match l2_bits:
        case 0:
            temp = y
            y = x
            x = temp
            return helper(hilb_idx>>2, curr_order*2, hcurve_order, x, y)
        case 1:
            y += mult_constant
            return helper(hilb_idx>>2, curr_order*2, hcurve_order, x, y)
        case 2:
            x += mult_constant
            y += mult_constant
            return helper(hilb_idx>>2, curr_order*2, hcurve_order, x, y)
        case 3:
            temp = y
            y = (mult_constant - 1) - x
            x = (mult_constant - 1) - temp
            x += mult_constant
            return helper(hilb_idx>>2, curr_order*2, hcurve_order, x, y)

def hilbex_to_cartesian(hilb_idx: int, hcurve_order: int) -> tuple:
    n1_hash = {0: (0, 0), 1: (0, 1), 2: (1, 1), 3: (1, 0)}

    idx = hilb_idx

    fnum = idx & 3
    coord = n1_hash[fnum]
    x = coord[0]
    y = coord[1]

    return helper(hilb_idx >> 2, 4, hcurve_order, x, y)
```

```python
import numpy as np
import os
from hilbert_curve import hilbertize_arr
import matplotlib.pyplot as plt
import random

b_train = []

fp_train = './Dataset/Dataset/Benign/Benign train/'

for file in os.listdir(fp_train):
    print(file)
    fp = fp_train
    bytes = np.array([f"{n:08b}" for n in open(fp + file, "rb").read()])
    for i in range(len(bytes)):
        bytes[i] = int(bytes[i], 2)

    bytes = bytes.astype(np.uint8)

    arr = hilbertize_arr(bytes)
    b_train.append(arr)
    # plt.imshow(arr)

plt.imshow(b_train[random.randint(0, len(b_train)-1)])
```
[136]

The image on the left is a recursive implementation of the hindex2xy algorithm to convert array indices to cartesian coordinates that lie on a hilbert curve of some order of magnitude. The image on the right is the script that was run to process each exe file in a subdirectory into an array of bytes and then into an matrix representing the hilbert curve with each element of the matrix being the integer value of the corresponding byte from the array of bytes.

These images were then saved to their respective folders and another script was run to read in the image data, apply transformations to decrease overfitting of the model, and create the train and test datasplit

```python
import numpy as np
import matplotlib.pyplot as plt
import cv2
import os
import random
from sklearn.model_selection import train_test_split

def create_img_dset(fp, label):
    dset = []
    for file in os.listdir(fp):
        curr_fp = fp + f'/{file}'
        img = cv2.imread(curr_fp)

        if random.uniform(0, 1) > 0.7:
            img = cv2.GaussianBlur(img, (3, 3), sigmaX=3)

        dset.append((img, label))

    return dset

b_train = create_img_dset('./b_train', 0) + create_img_dset('./b_train_extra', 0) + create_img_dset('./mbenign_imgs/', 0)
b_test = create_img_dset('./b_test', 0)
v_files = create_img_dset('./v_files_preprocessed/', 1)

b_train_labels = [0 for i in range(len(b_train))]
b_test_labels = [0 for i in range(len(b_test))]

dataset = np.array(b_train + b_test + v_files[:len(b_train) + len(b_test) - 2])

X_train, X_test, y_train, y_test = train_test_split(dataset[:, 0], dataset[:, 1], test_size=0.2, random_state=40)
```

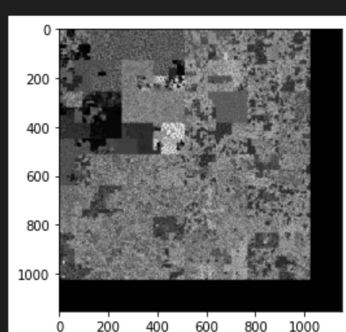
[1] ✓ 59.3s

To visualize images in the training set or any images from an array of data in the future the following code was used.

```python
plt.imshow(b_train[random.randint(0, len(b_train)-1)][0])
```

[2] ✓ 0.6s

```
<matplotlib.image.AxesImage at 0x22233adbb50>
```



The images were then put through a script that cropped out the excess blank parts of the image as can be seen in the rightmost and bottom columns of the visualized matrix in the above image.

```python
def cutoff_idx(arr: list, buffer) -> int:
    arr = list(arr)
    arr.reverse()

    for i in range(len(arr)):
        if sum(arr[i]) >= buffer:
            # if count > buffer:
            return len(arr) - 1 - i

def crop_img(img: np.ndarray, buffer: int) -> np.ndarray:
    # h_idx = cutoff_idx(img[0, :], buffer)
    # v_idx = cutoff_idx(img[:, 0], buffer)

    arrs = []
    for i in range(15):
        idx = random.randint(0, len(img[0]) // 2)
        if random.randint(0, 1) == 1:
            arrs.append(img[idx, :])
        else:
            arrs.append(img[:, idx])

    idxs = []
    for arr in arrs:
        cutoff = cutoff_idx(arr, buffer)
        if cutoff != None:
            idxs.append(cutoff)

    b_idx = max(idxs)

    return img[:b_idx, :b_idx]

img = crop_img(b_test[5][0], 5)
print(img.shape)
plt.imshow(img)
```

[7] ✓ 0.1s

```python
for i in range(len(X_train)):
    X_train[i] = crop_img(X_train[i], 5)
for i in range(len(X_test)):
    X_test[i] = crop_img(X_test[i], 5)
```

[8] ✓ 40.1s

After the images were cropped, all of them were rescaled or padded to reach the final shape of 64x64x3.

```python
def pad_img(img: np.ndarray, des_w: int) -> np.ndarray:
    while len(img[:, 0]) < des_w:
        row = np.zeros((1, len(img[0, :]), 3))
        img = np.vstack((img, row))
    while len(img[0, :]) < des_w:
        col = np.zeros((len(img[:, 0]), 1, 3))
        img = np.column_stack((img, col))

    return img

def rescale_img_arrs(arr):
    narr = []
    for i in range(len(arr)):
        if arr[i].shape[0] < 64:
            narr.append(pad_img(arr[i], 64))
        else:
            img = cv2.resize(arr[i], (64, 64))
            narr.append(img)

    return narr

nxtrain = rescale_img_arrs(X_train)
nxtest = rescale_img_arrs(X_test)
```

[10] ✓ 5.1s

After preprocessing all the images went through 3 feature extraction methods with the first being HOG feature extraction. HOG Feature extraction calculates the magnitude and orientation of the gradients of the images and builds a histogram from them.

$$G_x(x, y) = H(x + 1, y) - H(x - 1, y) \qquad (1)$$

$$G_y(x, y) = H(x, y + 1) - H(x, y - 1) \qquad (2)$$

$$G(x, y) = \sqrt{G_x(x, y)^2 + G_y(x, y)^2} \qquad (3)$$

$$\theta(x, y) = \arctan {}^{G_y(x, y)}\!/_{G_x(x, y)} \qquad (4)$$

```python
# HOG feature extraction
img = nxtrain[2]
plt.imshow(img)

x_grad_kernel = np.array([[ 0, 0, 0],
                          [-1, 0, 1],
                          [ 0, 0, 0]])

y_grad_kernel = np.array([[0, -1, 0],
                          [0,  0, 0],
                          [0,  1, 0]])

x_grad_kernel = np.array([[ 1, 0, 1],
                          [-2, 0, 2],
                          [ 1, 0, 1]])

y_grad_kernel = np.array([[1, -2, 1],
                          [0,  0, 0],
                          [1,  2, 1]])

def convolve_img(img: np.ndarray, kernel: np.ndarray) -> np.ndarray:
    return cv2.filter2D(img, -1, kernel)

def gradient_magnitude(x_grad: np.ndarray, y_grad: np.ndarray):
    return np.sqrt(x_grad**2 + y_grad**2)

def gradient_orientation(x_grad: np.ndarray, y_grad: np.ndarray):
    orientation_matrix = np.divide(y_grad, x_grad)
    orientation_matrix[np.isnan(orientation_matrix)] = 0

    return np.arctan(orientation_matrix)#  * (180 / np.pi)
```

```python
def find_closest_bin(val: int, bins: list) -> tuple:
    val = np.sum(val)//3

    # print(val)

    for i in range(len(bins)):
        if val == bins[i]:
            return bins[i]

    x1 = 0
    x2 = 20

    for i in range(len(bins)):
        if val > x1 and val < x2:
            return (x1, x2)

        x1 += 20
        x2 += 20

    min_dist = float('inf')
    closest_bin = None
    for i in range(len(bins)):
        dist = abs(val - np.sum(bins[i])//3)

        if dist < min_dist:
            closest_bin = bins[i]
            dist = min_dist

    if val > closest_bin:
        return (closest_bin, closest_bin + 20)
    elif val < closest_bin:
        return (closest_bin - 20, closest_bin)
    elif val == closest_bin:
        return closest_bin
```

```python
def bin_creation(mag_mat: np.ndarray, or_mat: np.ndarray) -> dict:
    bins = [i for i in range(0, 180, 20)]
    histogram = {}

    for bin in bins:
        histogram.update({bin: []})

    for i in range(len(mag_mat)):
        for j in range(len(mag_mat[i])):
            magnitude = np.sum(mag_mat[i][j])//3
            direction = np.sum(or_mat[i][j])//3

            # print(direction)

            closest_bin = find_closest_bin(direction, bins)

            if type(closest_bin) == tuple:
                lbound = closest_bin[0]
                ubound = closest_bin[1]

                udiff = ubound - direction
                ldiff = direction - lbound

                histogram[lbound].append((udiff/lbound)*magnitude)
                histogram[ubound].append((ldiff/lbound)*magnitude)
            elif type(closest_bin) == int:
                histogram[closest_bin].append(magnitude)

    return histogram

def extract_HOG_features(img: np.ndarray):
    x_grad, y_grad = convolve_img(img, x_grad_kernel), convolve_img(img, y_grad_kernel)
    magnitude_matrix = gradient_magnitude(x_grad, y_grad)
    orientation_matrix = gradient_orientation(x_grad, y_grad)

    histogram = bin_creation(magnitude_matrix, orientation_matrix)

    return histogram

histogram = extract_HOG_features(img)
print(histogram)
```

Local Binary Patterns was another feature extraction method used and labels pixels based on the intensity of surrounding pixels in comparison to an established center pixel. For each 3x3 patch in an image from the dataset the center of the patch would be compared to all directly surrounding 8 pixels. The surrounding pixels would be marked as either a 0 or 1 corresponding to whether they were lower than or greater than the center pixels respectively. Then a binary string of the surrounding pixels would be created starting from a pre-established surrounding pixel. The integer value of this binary string would be the new value of the center pixel.

```python
import numpy as np
import cv2
import matplotlib.pyplot as plt
import time

def get_hull(adder_pts: list, pt: tuple) -> np.ndarray:
    hull = []
    pti, ptj = pt
    for (i, j) in adder_pts:
        hull.append((pti + i, ptj + j))

    return hull

def get_points(arr: np.ndarray, pts: list) -> np.ndarray:
    arr_pts = []

    for (i, j) in pts:
        arr_pts.append(arr[i][j])

    return arr_pts

def extract_lbp(img: np.ndarray):
    adder_pts = [(-1, 1), (0, 1), (1, 1), (1, 0), (1, -1), (0, -1), (-1, -1), (-1, 0)]

    lbp_mat = np.zeros(img.shape)
    for i in range(1, len(img) - 1):
        for j in range(1, len(img) - 1):
            hull = get_hull(adder_pts, (i, j))
            barr = (get_points(img, hull) > img[i][j])*1

            num = int(''.join(map(str, barr)), base=2)
            lbp_mat[i][j] = num

    return lbp_mat

if __name__ == '__main__':
    fp = './hat_woman_LBP.jpg'

    img = cv2.imread(fp, 0)

    start = time.time()
    lbp_img = extract_lbp(img)
    print(f"finished in {time.time() - start}s")

    plt.imshow(lbp_img)
    plt.show()
```

The final method of feature extraction used was the convolution of the laplacian filter. The laplacian filter when convolved over the matrices in the dataset converts them into hessian matrices (matrices with second order partial derivatives as the elements). This filter was convolved to extract features over each image before CNN.

```python
import cv2
import numpy as np
import matplotlib.pyplot as plt
import time

def convolve_img(img: np.ndarray, kernel: np.ndarray) -> np.ndarray:
    return cv2.filter2D(img, -1, kernel)

laplacian_kernel = np.array([[-1, -1, -1],
                             [-1,  8, -1],
                             [-1, -1, -1]])

x_grad_kernel = np.array([[ 0, 0, 0],
                          [-1, 0, 1],
                          [ 0, 0, 0]])

y_grad_kernel = np.array([[0, -1, 0],
                          [0,  0, 0],
                          [0,  1, 0]])

img = cv2.imread('./lena.png')
img = nxtrain[0]

x_grad = convolve_img(img, x_grad_kernel)
y_grad = convolve_img(img, y_grad_kernel)

start = time.time()
lap_img = convolve_img(img, laplacian_kernel)
print(f"took {time.time() - start}s")

cv2.imwrite('lap_img.png', lap_img)

# plt.imshow(x_grad + y_grad)
plt.imshow(lap_img)
```

[12] ✓ 0.2s

After all 3 methods of feature extraction were complete, the CNN model was created using pytorch and the train loop began running using the features extracted as extra feature maps as input to the model.

```python
class CNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 4, 3, padding=1)
        self.conv2 = nn.Conv2d(4, 8, 2, padding=1)
        self.conv3 = nn.Conv2d(8, 16, 1, padding=0)
        self.pool = nn.MaxPool2d(2, 2)

        self.dropout = nn.Dropout(p=0.2)
        self.dropout2 = nn.Dropout(p=0.2)

        self.l1 = nn.Linear(1024, 256)
        self.l2 = nn.Linear(256, 128)
        self.l3 = nn.Linear(128, 64)
        self.l4 = nn.Linear(64, 32)
        self.out = nn.Linear(32, 1)
```

```python
    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool(x)
        x = self.dropout(x)

        x = F.relu(self.conv2(x))
        x = self.pool(x)
        x = self.dropout(x)

        x = F.relu(self.conv3(x))
        x = self.pool(x)
        x = self.dropout(x)

        x = torch.flatten(x)

        x = F.relu(self.l1(x))
        x = self.dropout2(x)

        x = F.relu(self.l2(x))
        x = self.dropout2(x)

        x = F.relu(self.l3(x))
        x = self.dropout2(x)

        x = F.relu(self.l4(x))
        x = self.dropout2(x)

        x = torch.sigmoid(self.out(x))

        return x
```

The train loop compiled 4 metrics to graph to evaluate the model performance during training including accuracy over the training set, loss over the training set, accuracy over a small part of the test set used as a validation accuracy, and loss over a small part of the test est used as a validation loss.

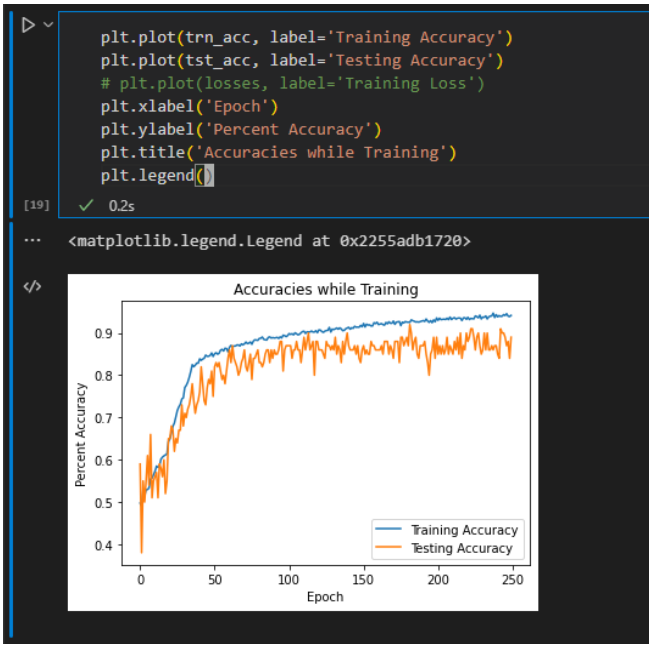
```python
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from LBP import extract_lbp
from skimage.feature import local_binary_pattern

def convolve_img(img: np.ndarray, kernel: np.ndarray) -> np.ndarray:
    return cv2.filter2D(img, -1, kernel)

laplacian_kernel = np.array([[-1, -1, -1],
                             [-1,  8, -1],
                             [-1, -1, -1]])

ksize = 5
sigma = 5
theta = 1*np.pi/4
lamda = 1*np.pi/4
gamma = 0.1
phi = 0

gabor_kernel = cv2.getGaborKernel((ksize, ksize), sigma, theta, lamda, gamma, phi, ktype=cv2.CV_32F)

radius = 3
num_points= 8 * radius

# model = models.resnet18(pretrained=True)# CNN()
model = CNN()

criterion = nn.BCELoss()
optimizer = optim.SGD(model.parameters(), lr=0.0001)# , momentum=0.9)
# optimizer = optim.Adam(model.parameters(),lr=0.0001,betas=(0.9,0.999),eps=1e-08,weight_decay=0,amsgrad=False)

losses = []
tst_losses = []
trn_acc = []
tst_acc = []

early_stopping = EarlyStopping(tolerance=10, min_delta=0.05)
```

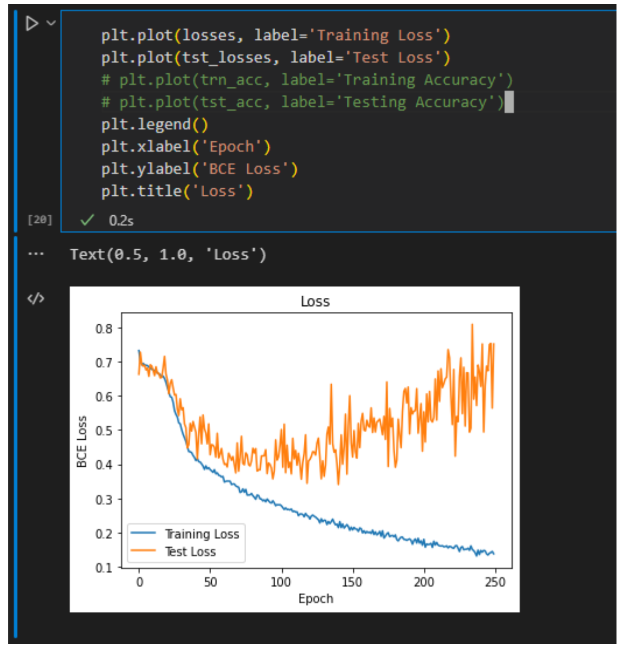
```python
BATCH_SIZE = len(nxtrain)
c1 = 0
c2 = BATCH_SIZE

reset_batch = False

min_loss = float('inf')
for epoch in range(250):
    closs = 0
    train_pred = 0
    for i in range(c1, c2, 1):
        preds = model(torch.Tensor([nxtrain[i][:, :, 0]]))

        if preds >= 0.5:
            pred = 1
        else:
            pred = 0

        if pred == y_train[i]:
            train_pred += 1

        # print(preds[0], y)

        loss = criterion(preds, torch.Tensor([y_train[i]]))

        closs += loss.item()

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    c1 += BATCH_SIZE
    c2 += BATCH_SIZE

    if c2 > len(nxtrain):
        c2 = len(nxtrain)
        reset_batch = True
```

```python
    # if closs/len(nxtrain) < min_loss:
    #     npath = './model_Gabor_Laplacian_HCURVE_dropout0_3_150epochs'
    #     torch.save(model.state_dict(), npath)
    #     min_loss = closs/len(nxtrain)
    #     print(f"Model Saved - Loss {min_loss}")

    pred_count = 0
    tmp_loss = 0
    for i in range(len(nxtest[:100])):
        pred = model(torch.Tensor([nxtest[i][:, :, 0]]))

        loss = criterion(pred, torch.Tensor([y_test[i]]))
        tmp_loss += loss.item()

        pred = pred.cpu().detach().numpy()[0]

        if pred >= 0.5:
            pred = 1
        else:
            pred = 0


        if pred == y_test[i]:
            pred_count += 1

    # trn_acc.append(train_pred/len(nxtrain))
    trn_acc.append(train_pred/BATCH_SIZE)
    losses.append(closs/BATCH_SIZE)
    tst_acc.append(pred_count/100)
    tst_losses.append(tmp_loss/100)
    # losses.append(closs/len(nxtrain))

    # early_stopping(losses[-1], tst_losses[-1])
    # if early_stopping.early_stop:
    #     print(f"EARLY STOPPED AT EPOCH {epoch}")
    #     break

    print(f"[ Epoch {epoch} | Train Acc {round(trn_acc[-1], 4)} Test Acc {round(tst_acc[-1], 4)} | TRN LOSS {round(losses[-1], 4)} TST LOSS {round(tst_losses[-1], 4)} ]")
    if reset_batch:
        c1 = 0
        c2 = BATCH_SIZE
```

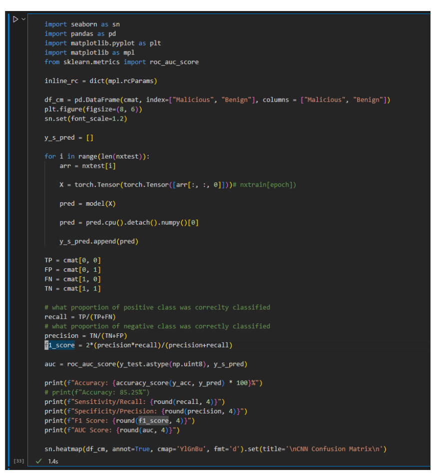
[46] 23m 18.8s Python

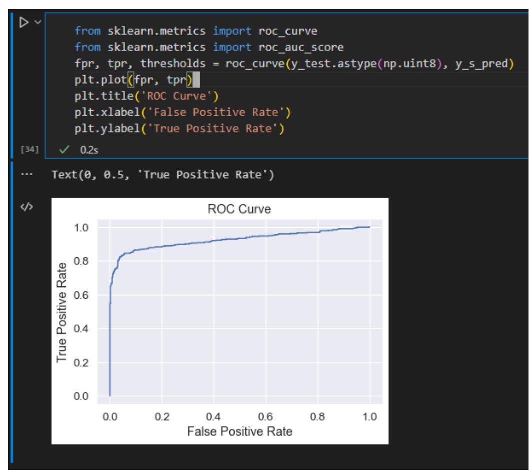
After training the model's accuracy on the train set and validation set as well as the model's loss on the train set and validation set are plotted. These can be analyzed to look for overfitting resulting in overcomplexity in the model (a high number of parameters and the possible memorization of portions of the traininset) or underfitting in the model (a high bias in the model where there are not enough parameters to classify the data well). During over fitting the validation loss will began to flatline or even increase slightly while training loss decreases significantly. Underfitting is characterized by a large close to constant gap occurring between most parts of the loss graph while both validation and training loss are decreasing





After visualizing the accuracy and loss, the model begins an evaluation process. First the model is set to evaluation mode which disables dropout regularization allowing all parameters that were learned to be utilized during inference. After calculating accuracy, a confusion matrix is calculated to visualize the faults of the model (where it classifies false positives and negatives) along with other metrics associated with a confusion matrix which can help to further evaluate the model. Finally the ROC curve is drawn (a standard metric that graphs the true positive rate agains the false positive rate). All metric calculated include accuracy, Sensitivty/Recall, Specificity/Precision, F1 Score, and AUC score.





After evaluation the model achieved an accuracy of 90.3191%, a Sensitivity/Recall of 0.8605, a Specificity/Precision of 0.9536, an F1 Score of 0.9047, and an AUC score of 0.9213. As these metrics approach 1 the model's accuracy, precision, and generalizability increase.

```
Accuracy: 90.31914893617021%
Sensitivity/Recall: 0.8605
Specificity/Precision: 0.9536
F1 Score: 0.9047
AUC Score: 0.9213

[Text(0.5, 1.0, '\nCNN Confusion Matrix\n')]
```