Binary Classification of images in Python

This notebook shows the workflow for building, training, and testing a custom model class using Pytorch for Machine learning. The specific purpose is Binary Classification of images.
The end goal is to create a model that will be suitable for scaling up as well as deployment.

The images can be downloaded from the following sources:
https://www.microsoft.com/en-us/download/details.aspx?id=54765
And
https://www.kaggle.com/datasets/samuelcortinhas/cats-and-dogs-image-classification/data?select=train

Training is implemented using Pytorch Ignite.

Project Overview:
The code in the cells of this notebook performs the following:

The images are first cleaned, duplicate checking is performed using the ImageHash package.
The data is visualized in various ways for inspection and inference.
The image files contained in the train/test/val folders have label files created which are stored in the main working directory.
The resulting file names will be:

[“training_labels_final.csv”, “val_labels_final.csv”, “test_labels_final.csv”]

A transformer is initialized. Which performs various types of data augmentation.

```
## Transformer
img_transforms = v2.Compose([v2.Resize((64,64)), ← Resizes the images.
                             v2.RandomHorizontalFlip(p=0.5),     ← Flips the images.
                             v2.RandomPhotometricDistort(p=0.5), ← Distorts the images.
```

```
                                   v2.ToTensor()])     ← Turns images into tensors.
```

Pytorch Custom Data set classes are then implemented. And have the following structure:

```
class Loading_training(Dataset):
    def __init__(self): ← Initializes the directory.
        """Loading various data sets
        but only in existing train, val test folders."""
    def __len__(self):  ← Provides the number of samples in dataset.
    def __getitem__(self, idx): ← Loads samples from a given index.
```

A data loader is then used to ‘feed’ the files to the model.

```
train_dataloader = DataLoader(training, batch_size=batch_size, shuffle=True)
```

A custom Neural Network model is initialized.

```
class Modelo(nn.Module): ← The model class. A neural network.
    def __init__(self):  ← Contains model layers.
        super(Modelo, self).__init__()
    def forward(self, x): ← Defines the order in which the layers are applied.
```

A loss function and optimizer are chosen.

```
loss_fn_ = nn.BCEWithLogitsLoss() ← Combines sigmoid layer with BCE Loss.
optimizer = torch.optim.AdamW(model.parameters(), lr = lr) ← Specifies learning rate
(and other params).
```

Pytorch Ignite is used for model training and to create model checkpoints.

```
def update_model(engine, batch): ← Creates a basic trainer.
    return x()
trainer = Engine(update_model)   ← Loops over the training data.

val_metrics = {                  ← Metrics to be tracked.
    "metric": metric(),
}
def validation_step(engine, batch): ← Runs model validation.
    return x()
evaluator = Engine(validation_step) ← Loops over the validation data.

@trainer.on(Events.ITER_COMPLETED(every=x)) ← How often to log events(iteration).
def log_training(engine):

@trainer.on(Events.EPOCH_COMPLTED(every=x)) ← How often to log events(per X epochs).
def run_validation():
    evaluator.run(val_dataloader)

def score_function(engine): ← Returns metrics that were defined in val_metrics.
    return engine.state.metrics

model_checkpoint = ModelCheckpoint("checkpoint") ← Creates a checkpoint.

trainer.run(train_dataloader, max_epochs=X) ← Runs the trainer.
```

The checkpoint models are then tested to see which performs best against unseen data from the original dataset as well as data from a completely new dataset.

```
for j in os.listdir(X) ← Iterates over all of the models.
    PATH=os.path.join(X,j)
    model.load_state_dict(torch.load(PATH)) ← Loads state dictionaries.
    for i,batch in enumerate(test_dataloader): ←Performs testing.
             . . .
```

Code:

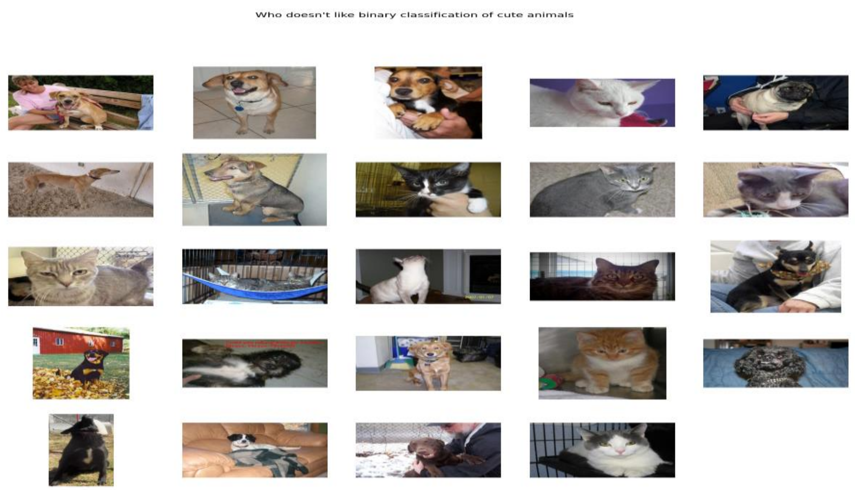
Installing libraries and dependencies.

Python version 11.3.7

```
#Installing libraries and dependencies

import os
import pandas as pd
import numpy as np
import torch
from ignite.handlers import Checkpoint
from PIL import Image
from torchvision.transforms import v2
from torch.utils.data import Dataset
from torchvision.transforms import ToTensor
from torchvision.io import read_image
from torch import nn
from torch.utils.data import DataLoader
from torchvision.transforms import Compose, Normalize, ToTensor
from ignite.engine import Engine, Events, create_supervised_trainer,
create_supervised_evaluator
from ignite.metrics import Accuracy, Loss
from ignite.handlers import ModelCheckpoint
from ignite.contrib.handlers import TensorboardLogger, global_step_from_engine
from ignite.handlers import Timer, BasicTimeProfiler, HandlersTimeProfiler
import warnings
warnings.filterwarnings('ignore')
```

Visualizations and clean up

```
larger_list=[]
directory = os.path.join(os.getcwd(), ‘train’)
q = 1
fig,ax = plt.subplots(figsize=(15, 15))
for I in range(0,(25)):
    random_file = random.choice(os.listdir(directory))
    abc = (os.path.join(directory, random_file)) #<- get the first file name
    picture = Image.open(abc) #<- reading those images
    plt.subplot(5,5,q) #<- you know that is so neat I am leaving it in
    plt.axis(‘off’)
    title = (“Who doesn’t like binary classification of cute animals”)
    plt.imshow(picture)
    plt.suptitle(title)
    ax.set_axis_off()
    q+= 1 #<- Make it so there are not so many files
    #print(q) #<- So I know it is not running forever
    if q == 25:
       break
```



Checking for duplicate values using hashing

The average_hash function was used from the Imagehash package.

https://pypi.org/project/ImageHash/

The image is first scaled down, then changed to grayscale, the means of the image colors are computed. The bits are then computed. A hash is then constructed by turning the 64 bits into a 64 bit integer. And if the results are the same, the images are likely the same. https://www.hackerfactor.com/blog/index.php?/archives/432-Looks-Like-It.html

The code below makes lists of any possible duplicates in the files. And then erases them.

```
list_for_hashing = []
files_to_keep = []
files_to_purge = []
files_to_purge1 = []
for file in os.listdir(directory1):
    image1=imagehash.average_hash(Image.open(os.path.join(directory1,file)))
    if image1 not in list_for_hashing:
        list_for_hashing.append(image1)
        files_to_keep.append(file)
    else:
        print(file,image1)
        files_to_purge.append([file,image1])
        files_to_purge1.append(image1)
```

```
zipped = list(zip(files_to_keep, list_for_hashing))
m=pd.DataFrame(zipped)
q = pd.DataFrame(files_to_purge)
pqm= q.merge(m, on = 1, how = 'inner')
file_list = list(pqm['0_x'])
file_list1 = list(pqm['0_y'])
```

This code plots all of the duplicate images

```
from matplotlib import pyplot as plt
larger_list=[]
directory= (os.getcwd()+'\PetImages\Cat')
q = 1
```

```
lengths1=len(pqm)
fig,ax = plt.subplots(figsize=(15,15))
for i in os.listdir(directory):
    if i in file_list:
        abc=(os.path.join(directory, i))
        picture = Image.open(abc) #<- reading those images
        plt.subplot(8,8,q) #<- you know that is so neat I am leaving it in
        plt.axis('off')
        title =("Who doesn't like binary classification of cute animals")
        plt.imshow(picture)
        plt.suptitle(title)
        ax.set_axis_off()
        q+=1 #<- Make it so there are not so many files
```

This portion of code erases the duplicates

```
directory= (os.getcwd()+'\train)
for file in os.listdir(directory):
    if file in file_list1:
        okay= os.path.join(directory,file)
        if os.path.exists(okay):
            os.remove(okay)
        else:
            pass
```

```
from matplotlib import pyplot as plt
larger_list=[]
directory= (os.getcwd()+'\train)
q = 1
lengths1=len(pqm)
fig,ax = plt.subplots(figsize=(15,15))
for i in os.listdir(directory):
    if i in file_list:
        abc=(os.path.join(directory, i))
        picture = Image.open(abc) #<- reading those images
        plt.subplot(5,5,q) #<- you know that is so neat I am leaving it in
        plt.axis('off')
        title =("Who doesn't like binary classification of cute animals")
        plt.imshow(picture)
        plt.suptitle(title)
        ax.set_axis_off()
        q+=1 #<- Make it so there are not so many files
```

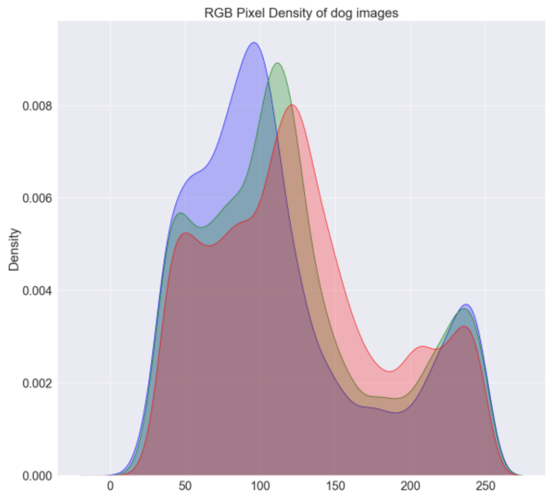
The result is a blank plot. So I will not show it here.

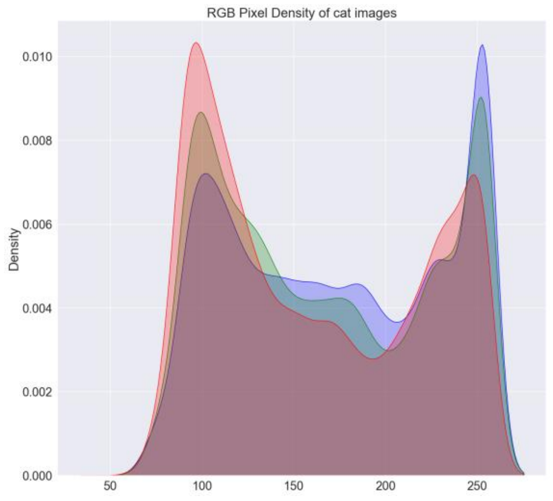
Making inferences about the image data, such as pixel density, the sizes of the images and the overall balance of the dataset.

```
pixel_values1 = []
widths_cat=[]
heights_cat=[]
widths_dog=[]
heights_dog=[]
traindir= directory
partial_string = 'og'
for img in os.listdir(traindir):
    if 'og' in img:
        img_path = os.path.join(traindir +'\\'+img ) # Making image file path
        im = Image.open(img_path)
        widths_dog.append(im.size[0])
        heights_dog.append(im.size[1])
        pixel_values1 = list(im.getdata())
    else:
        img_path = os.path.join(traindir +'\\'+img ) # Making image file path
        im = Image.open(img_path)
        widths_cat.append(im.size[0])
        heights_cat.append(im.size[1])
        pixel_values2 = list(im.getdata())
```

```
pixel_vals_b_dog = [x[0] for x in pixel_values1]
pixel_vals_g_dog = [x[1] for x in pixel_values1]
pixel_vals_r_dog = [x[2] for x in pixel_values1]
pixel_vals_b_cat = [x[0] for x in pixel_values2]
pixel_vals_g_cat = [x[1] for x in pixel_values2]
pixel_vals_r_cat = [x[2] for x in pixel_values2]
```

```
f,ax= plt.subplots(figsize=(16,15))
sns.kdeplot(pixel_vals_b_dog, ax=ax,color='blue',fill=True).set(title='RGB Pixel
Density of dog images')
sns.kdeplot(pixel_vals_g_dog, ax=ax,color='green', fill = True)
sns.kdeplot(pixel_vals_r_dog, ax=ax,color='red', fill =True)

f,ax= plt.subplots(figsize=(16,15))
sns.kdeplot(pixel_vals_b_cat, ax=ax,color='blue',fill=True).set(title='RGB Pixel
Density of cat images')
sns.kdeplot(pixel_vals_g_cat, ax=ax,color='green', fill = True)
sns.kdeplot(pixel_vals_r_cat, ax=ax,color='red', fill =True)
plt.show()
```





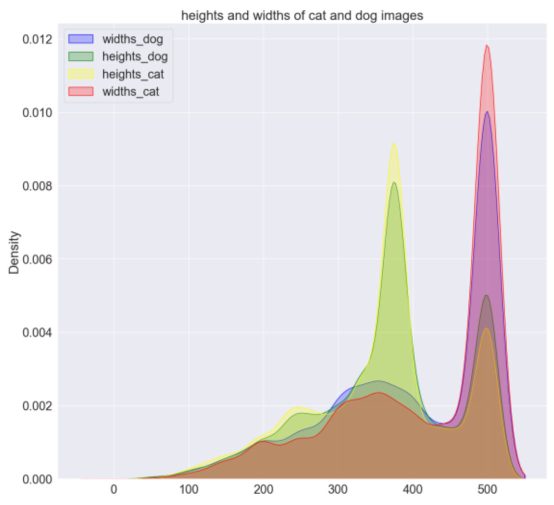
Images are composed of pixels. Pixels store information about their colors, in the form of 3 values. Red, green, blue. (R, G, B) When a package like imread is used to process an image, the resulting data is stored in a tuple (list?) where the first value is R, the second is G, the third is B. A plot of values from each category can provide insight about the general composition of images.

```
# Pixel values need to be in range 0/1
pixel_values1 = [50/255, 50/255, 50/255]
pixel_values2 = [100/255, 100/255, 100/255]
pixel_values3 = [150/255, 150/255, 150/255]
def plot_pixels(pixel_values):
    patch = plt.Rectangle((0, 0), 1, 1, color = pixel_values)
    fig, ax = plt.subplots()
    ax.add_patch(patch)
    ax.set_xlim(0, 1)
    ax.set_ylim(0, 1)
    ax.axis('off')
plot_pixels(pixel_values1), plot_pixels(pixel_values2), plot_pixels(pixel_values3)
plt.show()
```

There is a lot of density in the 50, 100, and 150 range for both sets of images. And when the colors are plotted, voilà. Many of the images contain large proportions of neutral colors.

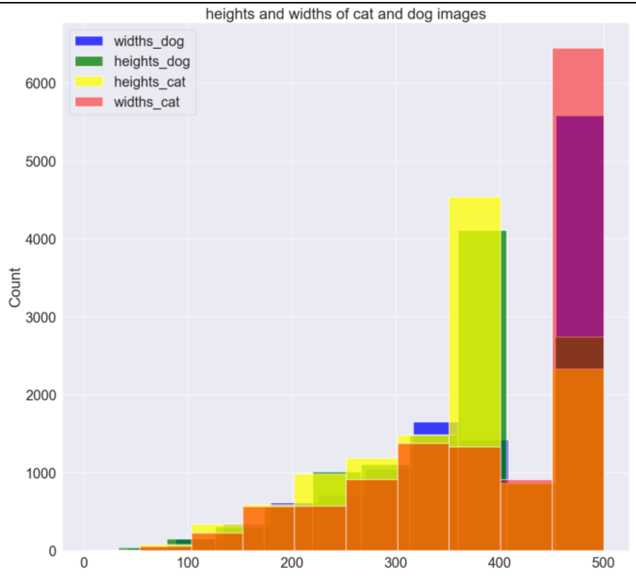
What about size distributions between the images? Are dog or cat images generally larger?

```
f,ax= plt.subplots(figsize=(16,15))
sns.kdeplot(widths_dog, ax=ax,color='blue',fill=True).set(title=
'heights and widths of cat and dog images')
sns.kdeplot(heights_dog, ax=ax,color='green', fill = True)
sns.kdeplot(heights_cat, ax=ax,color='yellow', fill = True)
sns.kdeplot(widths_cat, ax=ax,color='red', fill = True)
plt.legend(['widths_dog','heights_dog','heights_cat','widths_cat'],loc='upper left')
plt.show()
```



Most images for both categories are generally on the larger side. Most heights cluster around 350 pixels for both cat and dog images. While many of the images are about 500 pixels high.

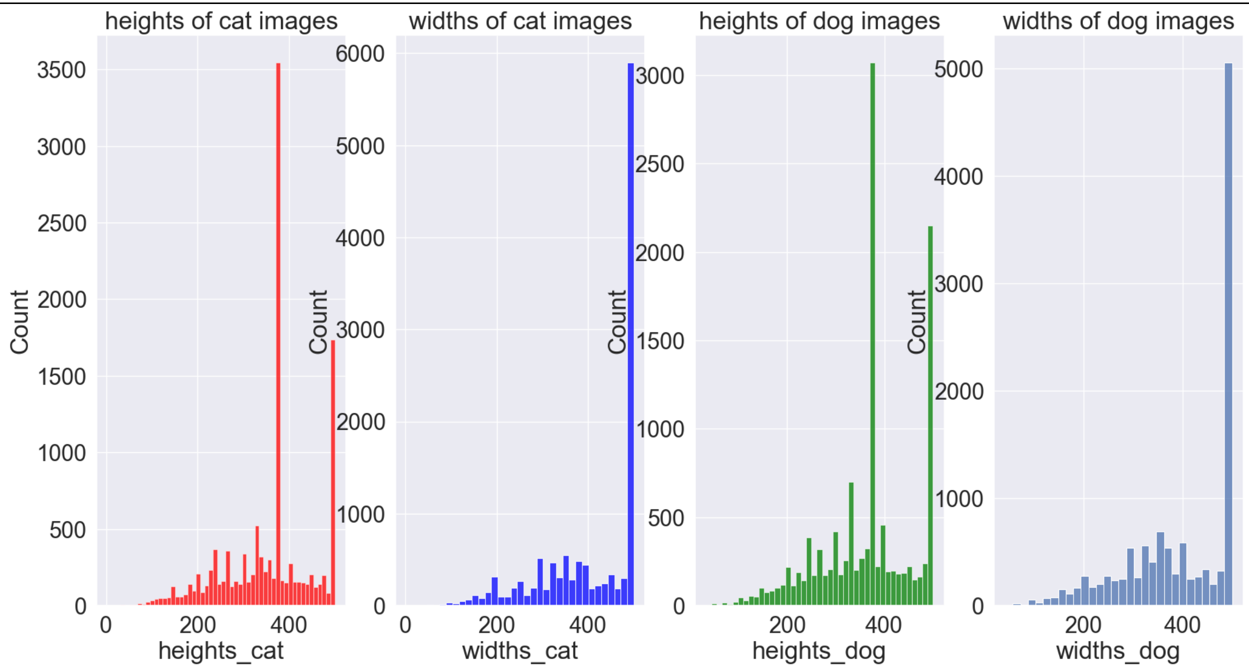
```
f,ax= plt.subplots(figsize=(16,15))
sns.histplot(widths_dog, ax=ax,color='blue',fill=True, bins =10).set(title=
'heights and widths of cat and dog images')
sns.histplot(heights_dog, ax=ax,color='green', fill = True,bins=10)
sns.histplot(heights_cat, ax=ax,color='yellow', fill = True,bins=10)
sns.histplot(widths_cat, ax=ax,color='red', fill = True,alpha= 0.5,bins=10)
plt.legend(['widths_dog','heights_dog','heights_cat','widths_cat'],loc='upper left')
plt.show()
```



The same data visualized as a histogram shows much the same pattern. Almost half of the 24,000 images are 500 pixels wide. A set of subplots helps illustrate this better.

```
f,axes = plt.subplots(1, 4,figsize=(20,10))
cats_heights1 = pd.DataFrame(heights_cat,columns=['heights_cat'])
cats_width1 = pd.DataFrame(widths_cat,columns=['widths_cat'])
dogs_heights1=pd.DataFrame(heights_dog,columns=['heights_dog'])
dogs_width1=pd.DataFrame(widths_dog,columns=['widths_dog'])
```

```
sns.histplot(cats_heights1, x="heights_cat",ax=axes[0],
color='red').set(title="heights of cat images")
sns.histplot(cats_width1, x="widths_cat",ax=axes[1],color='blue').set(title="widths
of cat images")
sns.histplot(dogs_heights1,
x="heights_dog",ax=axes[2],color='green').set(title="heights of dog images")
sns.histplot(dogs_width1, x="widths_dog",ax=axes[3]).set(title="widths of dog
images")
```



The subplots help illustrate the spread of image sizes. All sets of images have one category that is as large as the sum of all of the others. The largest categories for heights are the same for both cat and dog images. About 400 pixels and 500 respectively. And both have similar counts for each, about 3500 images at about 400 pixels high and about 2000 at 500 pixels for cats. Versus about 3000 images at 400 pixels and about 2000 images at 500 pixels for dogs.
Widths exibit a similar pattern, both cat and dog images cluster at about 500 pixels for the most frequent category. With cat images at this size numbering almost 6000, and dog images trailing closely behind at about 5000.

What about the balance of samples? How many classes are found in each data set?

Since the problem is binary, the images fall into categories of 0 or 1.

```
print(' There are:', len(xy1['status'].unique()), 'categories in the training data set','\n',
'There are:', len(xy2['status'].unique()), 'categories in the validation data set','\n',
'There are:', len(xy3['status'].unique()), 'categories in the test data set')
```

```
There are: 2 categories in the training data set
There are: 2 categories in the validation data set
There are: 2 categories in the test data set
```

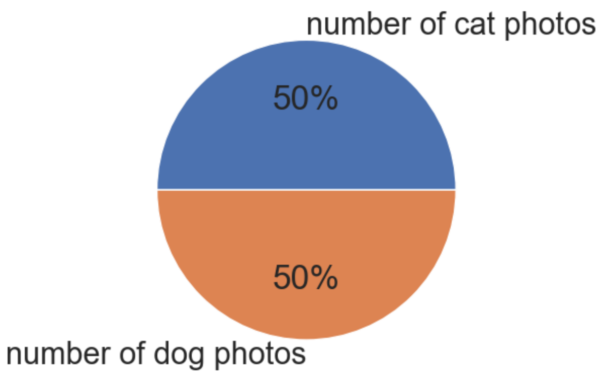
Not surprisingly, there are two categories in each data set.
How about the numbers of samples?

```
print('Training data:', 'images in the 0 category : ',len(xy1[xy1['status']==0]),","
      ,'images in the 1 category :',len(xy1[xy1['status']==1]) )
print('Validation data:', 'images in the 0 category:',len(xy2[xy2['status']==0]),","
      ,'images in the 1 category:',len(xy2[xy2['status']==1]) )
print('Test data:', 'images in the 0 category:',len(xy3[xy3['status']==0]),","
      ,'images in the 1 category:',len(xy3[xy3['status']==1]) )
```

```
Training data: images in the 0 category :  12418 , images in the 1 category : 12418
Validation data: images in the 0 category: 40 , images in the 1 category: 40
Test data: images in the 0 category: 40 , images in the 1 category: 40
```

To help drive home that the data set contains equal numbers. A pie chart is included.

```
data=[len(heights_cat), len(heights_dog)]
keys='number of cat photos','number of dog photos'
palette_color = sns.color_palette('light')
plt.pie(data, labels=keys, colors = palette_color,autopct='%.0f%%')
plt.show()
```



Unsurprisingly, the pie is split evenly.

Procedure

Setting the device to cuda, so the GPU can be utilized.

```
print(torch.__version__)
device = torch.device('cuda')
```

Creating lists, one for training, validation and testing, appending relevant file names to each.

```
# Loading data
# For training

starter_path = os.path.join(os.getcwd(), 'train')
data_labels_tr = []
next_path = os.listdir(starter_path)

for i in next_path:
    impath = (os.path.join(starter_path, i))
    data_labels_tr.append(i)

#For validation
```

```python
starter_path = os.path.join(os.getcwd(), 'val')
data_labels_val = []
next_path = os.listdir(starter_path)

for i in next_path:
    impath = (os.path.join(starter_path, i))
    data_labels_val.append(i)

#For testing
starter_path = os.path.join(os.getcwd(), "test")
data_labels_tst = []
next_path = os.listdir(starter_path)

for i in next_path:
    impath = (os.path.join(starter_path, i))
    data_labels_tst.append(i)
```

Creating labels by matching image names to partial strings. If the image contains dog or in this case 'og' a 1 will be added to a new column titled 'status'. Column names are also added to simplify my life.

```python
# A lazyish way of creating labels for a binary classification task. Put a 0 if the #
# file contains a partial name
partial_string = 'og'
xy1=pd.DataFrame(data_labels_tr)
xy2=pd.DataFrame(data_labels_val)
xy3=pd.DataFrame(data_labels_tst)
xy1['status'] = xy1[0].str.contains(partial_string).astype(int)
xy2['status'] = xy2[0].str.contains(partial_string).astype(int)
xy3['status'] = xy3[0].str.contains(partial_string).astype(int)
xy1.columns=['id','status']
xy2.columns=['id','status']
xy3.columns=['id','status']
```

All data is saved to csv files, which **will** overwrite whatever is in the directory if it exists.

```
# Save to csv files in the current directory
xy1.to_csv("training_labels_final.csv",index=False)
xy2.to_csv("val_labels_final.csv",index=False)
xy3.to_csv("test_labels_final.csv",index=False)
```

Resizing images, flipping them and turning them into tensors.

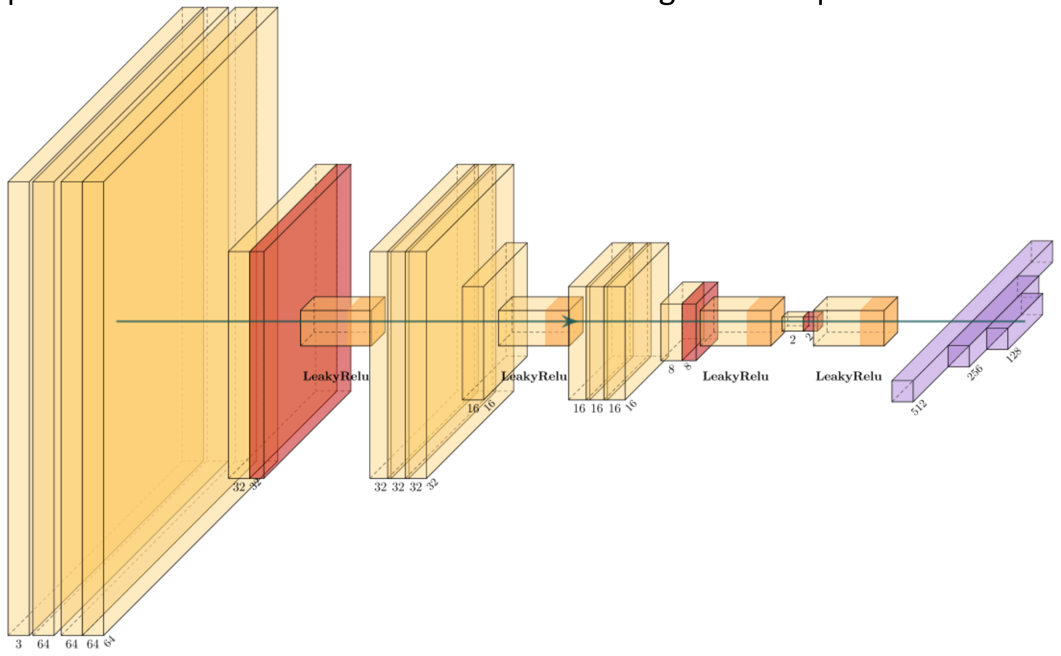
```
## Transformer
img_transforms = v2.Compose([v2.Resize((64,64)),
                             v2.RandomHorizontalFlip(p=0.5),
                             v2.RandomPhotometricDistort(p=0.5),
                            v2.ToTensor()])
```

Custom Data loaders load all labels, pair them to images and apply transformations.

```
# Custom data loaders for training validation testing
# Loads all images and puts them into either lists or into data dictionaries.

class Loading_training(Dataset):

    def __init__(self):
        """Loading various data sets
        but only in existing train, val test folders."""
        self.selected_dataset_dir = os.path.join(os.path.join(os.getcwd(), "train"))
        self.all_filenames = os.listdir(self.selected_dataset_dir)
        self.all_labels = pd.read_csv(os.path.join(os.getcwd(),
'training_labels_final.csv'))
        self.label_meanings = self.all_labels.columns.values.tolist()


    def __len__(self):
        """Weird"""
        return len(self.all_filenames)
    def __getitem__(self, idx):
        img_path = os.path.join(self.selected_dataset_dir, self.all_labels.iloc[idx,
0])
        image = Image.open(img_path).convert("RGB")
        label = self.all_labels.iloc[idx, 1]
        label = torch.tensor(label)
        image = img_transforms(image)
        return image, label


class Loading_val(Dataset):

    def __init__(self):
        """Trying to load jpeg transformed files."""
        self.selected_dataset_dir = os.path.join(os.getcwd(), "val")
        self.all_filenames = os.listdir(self.selected_dataset_dir)
        self.all_labels = pd.read_csv(os.path.join(os.getcwd(),
'val_labels_final.csv'))
        self.label_meanings = self.all_labels.columns.values.tolist()


    def __len__(self):
        """Weird"""
        return len(self.all_filenames)


    def __getitem__(self, idx):
        img_path = os.path.join(self.selected_dataset_dir, self.all_labels.iloc[idx,
0])
        image =  Image.open(img_path).convert("RGB")
        label = self.all_labels.iloc[idx, 1]
        label = torch.tensor(label)
        image = img_transforms(image)
        return image, label
```

```
class Loading_test(Dataset):

    def __init__(self):
        """Loading test images."""
        self.selected_dataset_dir = os.path.join(os.getcwd(), "test")
        self.all_filenames = os.listdir(self.selected_dataset_dir)
        self.all_labels =
pd.read_csv(os.path.join(os.getcwd(),'test_labels_final.csv'))
        self.label_meanings = self.all_labels.columns.values.tolist()


    def __len__(self):
        """Weird"""
        return len(self.all_filenames)


    def __getitem__(self, idx):
        """The IDX function is built in to Pytorch. Pretty Cool I think.
        And then I use the above image transforms function"""
        selected_filename = self.all_filenames[idx]
        #print(selected_filename)
        imagepil = Image.open(os.path.join(self.selected_dataset_dir,
selected_filename)).convert("RGB")
        image = img_transforms(imagepil)
        self.all_labels = pd.read_csv(os.path.join(os.getcwd(),
'test_labels_final.csv'))
        label= self.all_labels['status'][idx]


        sample = {'data':image,
                  'label':label,
                  'img_idx':idx, 'sample_name':selected_filename}
        return(sample)
```

The model accepts 64 X 64 3 channel RGB image as input, and casts a prediction about the class of the image as output.



Several convolutional layers are stacked together at a time into blocks. The consistent padding throughout the model maintains sizes.
Layer size can be calculated using the following formula:

$$\frac{I-K+(2*P)}{S}+1$$

- Where I is the shape of your input.
- K is the filter size. (The kernel size)
- P is the padding of the input.
- S is the stride.

The first input size is 64 X 64 X 3, and this shape is maintained for almost the entire block.

A calculation for the first layer shape as an example:

$$\frac{64-3+(2*1)}{1}+1=64$$

The second block scales outputs to 32 X 64 X 3 then further down to 32 X 32 X 3. The third block, to 16 X 16 X 3 and the final block uses

dimensions of 8 X 8 X 3. With a single final layer to scale the output down to an appropriate number of channels (2).
Within the model layers MaxPooling is also used for scaling down inputs. Each application reduces the size by a factor of 2.
First MaxPool = 64 / 2 = 32
Second MaxPool = 32 / 2 = 16
LeakyRelu is used throughout the model architecture. It is thought that LeakyRelu helps to combat the vanishing gradient problem. The final layers of the model are linear.
The first input is (2 X 16 X 16) or 512 features. Which is then divided by 2 and then further subdivided by 2. The final layer outputs the prediction.

First Linear Layer → 512 features → 256 features
Second Linear Layer → 256 features → 128 features
Third Linear Layer → 128 features → 1 feature
The loss function is Binary Cross Entropy with Logits Loss.
The Optimizer is AdamW.

Model Architecture:

```
class Modelo(nn.Module):


    def __init__(self):
        super(Modelo, self).__init__()
        self.main = torch.nn.Sequential(
            torch.nn.Conv2d(in_channels=3, out_channels=64, kernel_size=(3, 3),
padding=1),
            torch.nn.Conv2d(in_channels=64, out_channels=64, kernel_size=(3, 3),
padding=1),
            torch.nn.Conv2d(in_channels=64, out_channels=64, kernel_size=(3, 3),
padding=1),
            torch.nn.Conv2d(in_channels=64, out_channels=64, kernel_size=(3, 3),
padding=1),
            #torch.nn.Dropout2d(p=0.2),
            torch.nn.Conv2d(in_channels=64, out_channels=32, kernel_size=(3, 3),
padding=1,bias=False),
            nn.BatchNorm2d(32, affine=False),
            torch.nn.MaxPool2d(2, 2),
            torch.nn.LeakyReLU(inplace=True,negative_slope=0.02))
        self.main1 = torch.nn.Sequential(
        torch.nn.Conv2d(in_channels=32, out_channels=32, kernel_size=(3, 3),
padding=1),
        torch.nn.Conv2d(in_channels=32, out_channels=32, kernel_size=(3, 3),
padding=1),
        torch.nn.Conv2d(in_channels=32, out_channels=32, kernel_size=(3, 3),
padding=1),
        torch.nn.Dropout2d(p=0.2),
        torch.nn.Conv2d(in_channels=32, out_channels=16, kernel_size=(3, 3),
padding=1),
        torch.nn.LeakyReLU(inplace=True,negative_slope=0.02))
        self.main2 = torch.nn.Sequential(
        torch.nn.Conv2d(in_channels=16, out_channels=16, kernel_size=(3, 3),
padding=1),
        torch.nn.Conv2d(in_channels=16, out_channels=16, kernel_size=(3, 3),
padding=1),
        torch.nn.Conv2d(in_channels=16, out_channels=16, kernel_size=(3, 3),
padding=1),
        torch.nn.Conv2d(in_channels=16, out_channels=8, kernel_size=(3, 3),
padding=1),
        #torch.nn.MaxPool2d(2, 2),
        torch.nn.LeakyReLU(inplace=True,negative_slope=0.02))
        self.main3 = torch.nn.Sequential(
        torch.nn.Dropout2d(p=0.2),
        torch.nn.Conv2d(in_channels=8, out_channels=2, kernel_size=(3, 3),
padding=1),
        torch.nn.MaxPool2d(2, 2),
        torch.nn.LeakyReLU(inplace=True,negative_slope=0.1))
        #self.fc1 = nn.Linear(in_features=8192,out_features=4096)
        #self.fc2 = nn.Linear(in_features=4096,out_features=2048)
        #self.fc3 = nn.Linear(in_features=2048,out_features=1024)
        self.fc4 = nn.Linear(in_features=512,out_features=256)
        self.fc5 = nn.Linear(in_features=256,out_features=128)
        self.fc6 = nn.Linear(in_features=128,out_features=1)

    def forward(self, x):
        x = self.main(x)
        x = self.main1(x)
        x = self.main2(x)
        x = self.main3(x)
        x = torch.flatten(x,start_dim=1,end_dim=-1)
        #x = self.fc1(x)
        #x = self.fc2(x)
        #x = self.fc3(x)
        x = self.fc4(x)
        x = self.fc5(x)
        x = self.fc6(x)
        return x


model = Modelo().to(device)




training1=Loading_training()
val1=Loading_val()
test1= Loading_test()


train_dataloader = DataLoader(training1, batch_size=64, shuffle=True)
val_dataloader = DataLoader(val1, batch_size=10, shuffle=True)
```

```
loss_fn_=nn.BCEWithLogitsLoss()
optimizer = torch.optim.AdamW(model.parameters(), lr=0.0000050)
```

The trainer below is adapted from Pytorch Ignite.

```
def update_model(engine, batch):
    model.train()
    data,label = batch
    optimizer.zero_grad()
    data=data.to(device)
    label=label.to(device)
    #print(label.get_device())
    #print(data.get_device())
    outputs,_=(model(data),label)
    outputs=outputs.squeeze()
    loss = loss_fn_(outputs, label.float())
    loss.backward()
    optimizer.step()
    return loss.item()

trainer = Engine(update_model)


val_metrics = {
    "accuracy": Accuracy(),
    "loss": Loss(loss_fn_)
}


def validation_step(engine, batch):
    model.eval()
    with torch.no_grad():
        x, y = batch
        x=x.to(device)
        y=y.to(device)
        outputs,_=(model(x),y)
        outputs = outputs.squeeze()
        outputs=torch.sigmoid(outputs)
        outputs=outputs.round()
        y=y.cpu().detach()

    return outputs, y.float()
```

```
evaluator = Engine(validation_step)


from ignite.metrics import Accuracy
from ignite.metrics import Precision, Recall

precision = Precision()


Accuracy().attach(evaluator, "accuracy")
Precision().attach(evaluator,'precision')
Recall(average='weighted').attach(evaluator,'recall')




@trainer.on(Events.ITERATION_COMPLETED(every=100))
def log_training(engine):
    batch_loss = engine.state.output
    lr = optimizer.param_groups[0]['lr']
    e = engine.state.epoch
    n = engine.state.max_epochs
    i = engine.state.iteration
    print("Epoch {}/{} : {} - batch loss: {}, lr: {}".format(e, n, i, batch_loss,
lr))


from ignite.handlers import Timer, BasicTimeProfiler, HandlersTimeProfiler
from ignite.engine import Events

validate_every = 5


@trainer.on(Events.EPOCH_COMPLETED(every=validate_every))
def run_validation():
    evaluator.run(val_dataloader)

@trainer.on(Events.EPOCH_COMPLETED(every=validate_every))
def log_validation():
    ugh=[]
    metrics = evaluator.state.metrics
    print(f"Epoch: {trainer.state.epoch},  Accuracy: 
{metrics['accuracy']},  Precision: {metrics['precision']}, recall: 
{metrics['recall']}")


def score_function(engine):
    return engine.state.metrics["accuracy"]


model_checkpoint = ModelCheckpoint(
    "checkpoint",
    n_saved=25,
    filename_prefix="best",
    score_function=score_function,
    score_name="accuracy",
    global_step_transform=global_step_from_engine(trainer),
)

evaluator.add_event_handler(Events.COMPLETED, model_checkpoint, {"model": model})

tb_logger = TensorboardLogger(log_dir="tb-logger")

tb_logger.attach_output_handler(
    trainer,
    event_name=Events.ITERATION_COMPLETED(every=100),
    tag="training",
    output_transform=lambda loss: {"batch_loss": loss},
)

for tag, evaluator in [("training", trainer), ("validation", evaluator)]:
    tb_logger.attach_output_handler(
        evaluator,
        event_name=Events.EPOCH_COMPLETED,
        tag=tag,
        metric_names="all",
        global_step_transform=global_step_from_engine(trainer),
    )


trainer.run(train_dataloader, max_epochs=200)
```

```
Model iterating:

Epoch 1/200 : 100 - batch loss: 0.6717314720153809, lr: 5e-06
Epoch 1/200 : 200 - batch loss: 0.62602299451828, lr: 5e-06
Epoch 1/200 : 300 - batch loss: 0.5983356833457947, lr: 5e-06
Epoch 2/200 : 400 - batch loss: 0.620399534702301, lr: 5e-06
Epoch 2/200 : 500 - batch loss: 0.6466283798217773, lr: 5e-06
Epoch 2/200 : 600 - batch loss: 0.6703633069992065, lr: 5e-06
Epoch 2/200 : 700 - batch loss: 0.6332987546920776, lr: 5e-06
Epoch 3/200 : 800 - batch loss: 0.5965307950973511, lr: 5e-06

Epoch 3/200 : 900 - batch loss: 0.5554847121238708, lr: 5
```

Results and conclusion:

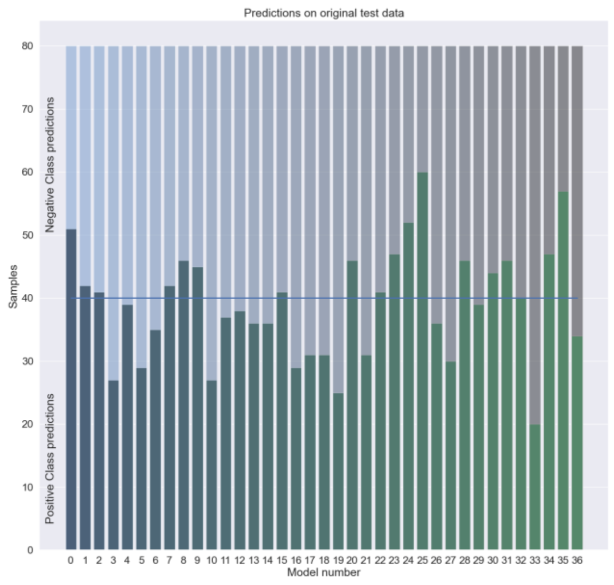
The performance of a machine learning model is often assessed using several metrics:
Accuracy, F1, Recall, Precision. The sections below provide inference about model performance in each. And how each is calculated involves gathering several types of results.

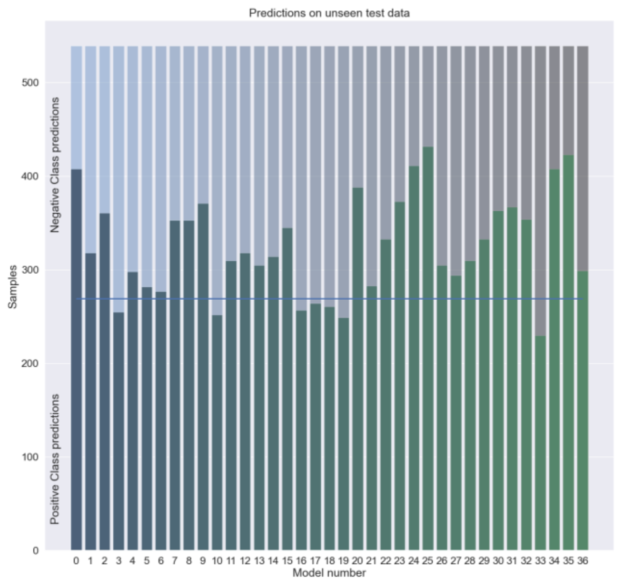
- True Positives = All results that are labelled 1 and are classified as 1 by the model.
- True Negatives = All results that are labelled 0 and are classified as 0 by the model.
- False Positives = All results that are labelled 0 but are classified as 1 by the model.
- False Negatives = All results that are labelled 1 but are classified as 0 by the model.

Often the results will be displayed in something called a confusion matrix. Which neatly displays the results of each category. The table is meant to be read left to right.
At the intersection of column 0 and row 0 are the truly negative results (TN). At the intersection of column 0 and row 1 are the false

negative results (FN). Column 1 and row 0 yields the false positive (PF) results. Lastly, the intersection of column 1 and row 1 shows the true positive (TP) results.

		Col. 0 (Label 0)	Col.1 (Label 1)
		0	1
Row 0 (Prediction 0)	0	TN	FP
Row 1 (Prediction 1)	1	FN	TP





Since there are so many models to iterate over the above figures help summarize the confusion matrices somewhat that were generated during the experiment. The table of confusion matrices can be found in the supplemental section of this report.

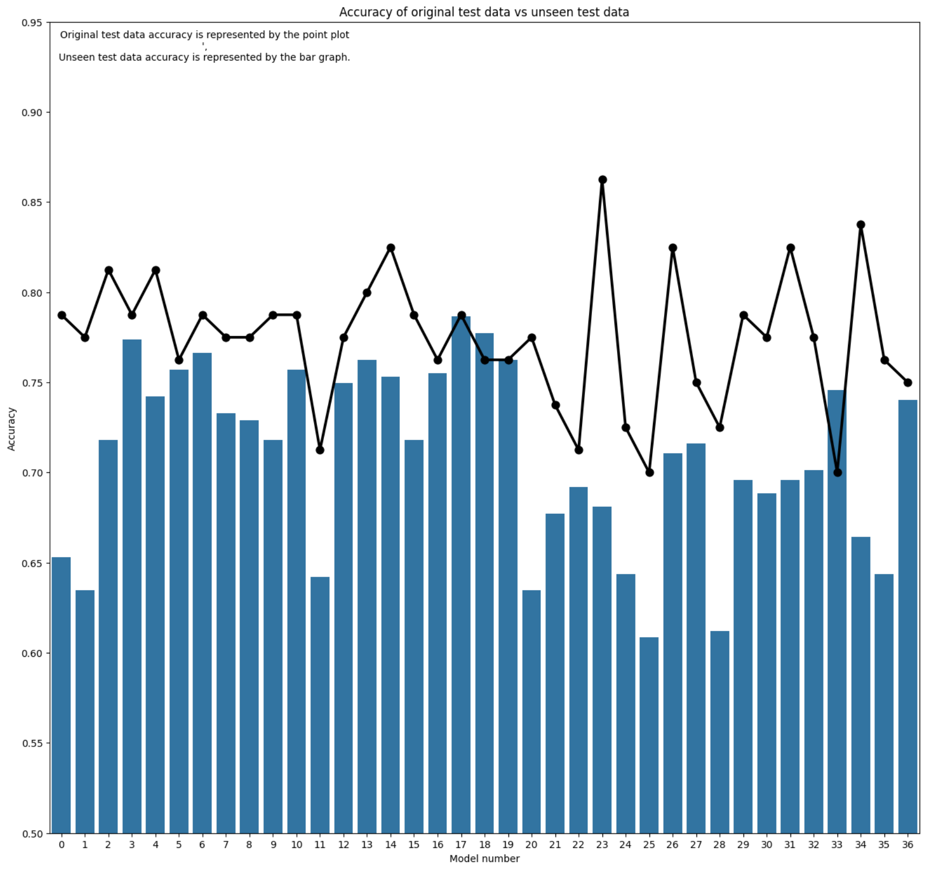
The figures are meant to be interpreted in the following way: Up to the blue line would be considered true positives. Above the blue line are true negatives. If either crosses the threshold, the result can be considered a False positive or False negative. For the original test data (Left figure) most models predicted more strongly toward the negative class. Overall there were many more false negatives than for the unseen data(out of sample data). While for the unseen data (Right figure) the opposite holds true. There are many more false postives and the proportion of true positives is also higher.

And when these results are taken and analyzed, depending on which are used the resulting metrics fall into Accuracy, F1, Recall, Precision.

Accuracy:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

- TP = The number of predictions that are labelled 1 and classified as 1 by the model.
- TN = The number of predictions that are labelled 0 and classified as 0 by the model.
- FP = The number of predictions that are labelled 0 and classified as 1 by the model.
- FN = The number of predictions that are labelled 1 and classified as 0 by the model.



Comparing initial accuracy and accuracy on completely new data.

The above figure shows that the model performed better on the original test data vs the unseen test data in all but a few cases.
Statistical testing is used to determine whether the original testing data accuracy is indeed better than the unseen data testing accuracy.
Modules from the Scipy.Stats package are used to perform the analysis.

Null Hypothesis:

$H_0 : \mu_1 = \mu_2$

Alternative Hypothesis:

$H_1 : \mu_1 \neq \mu_2$

Normality is tested for using the Shapiro-Wilk test. The formula for the Shapiro Wilk test is as follows:

$$w = \frac{\left(\sum_{i=1}^{n} a_i x_{(i)}\right)^2}{\sum_{i=1}^{n}(x-\bar{x})^2}$$

- x_i = the sample values
- a_i = mathematical constants that are calculated from a combination of the covariance matrix and expected values.

The scipy.stats.shapiro() test is used for this task.
The resulting p-values of 0.92 and 0.13 are greater than the assumed α of 0.05. So both sets of samples were assumed to come from a normal distribution.

Next the assumption of equal variance is tested using Levene’s Test.

The basic premise is that all samples have the same variance throughout. The null hypothesis is that samples all have equal variances.

Null Hypothesis:

$H_0 : \sigma_1^2 = \sigma_2^2 = \sigma_k^2$

The alternative Hypothesis is that samples do not have equal variances.

$H_1 : \sigma_1^2 \neq \sigma_k^2$

The scipy.stats.levene() test is used to perform this step. The resulting p-value of 0.03265 rejects the null hypothesis that all samples have equal variances.

So having samples that come from normal distributions that do not have equal variances, the two sample independent t-test with unequal variance is chosen (Welch's T-Test). The α value is 0.05.
The formula for Welch's T-Test is as follows:

$$t = \frac{\bar{x}_1 - \bar{x}_2}{\sqrt{s_{\bar{x}_1}^2 + s_{\bar{x}_2}^2}}$$

- $\bar{x}$ is the mean of each population
- $s_{\bar{x}_1}^2$ is the standard deviation of each population

The resulting p value of 0.0000000251 caused the rejection of the null hypothesis that the sample means were equal.

$Reject: H_0 : \mu_1 = \mu_2$

$In\, favor\, of : H_0 : \mu_1 \neq \mu_2$

A one tailed version of the Welch's t-test was subsequently performed to see if the accuracy of models tested on the original data was higher than models tested on unseen data. The α value is 0.05.

Null Hypothesis:

$H_0 : \mu_1 \geq \mu_2$

Alternative Hypothesis:

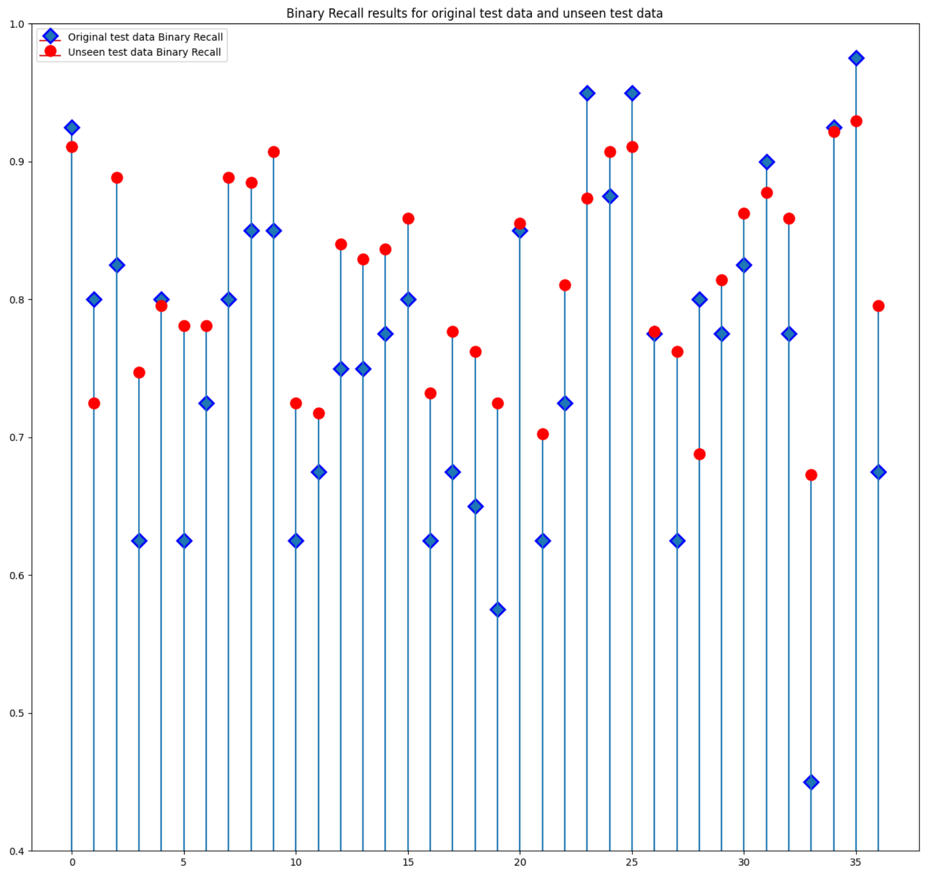
$H_1 : \mu_1 > \mu_2$

The resulting p-value of 0.0000000125 caused the rejection of the null hypothesis. The accuracy results of the original test data are indeed higher.

Binary Recall data:
The calculation for Binary Recall is:

$$Recall = \frac{TP}{TP + FN}$$

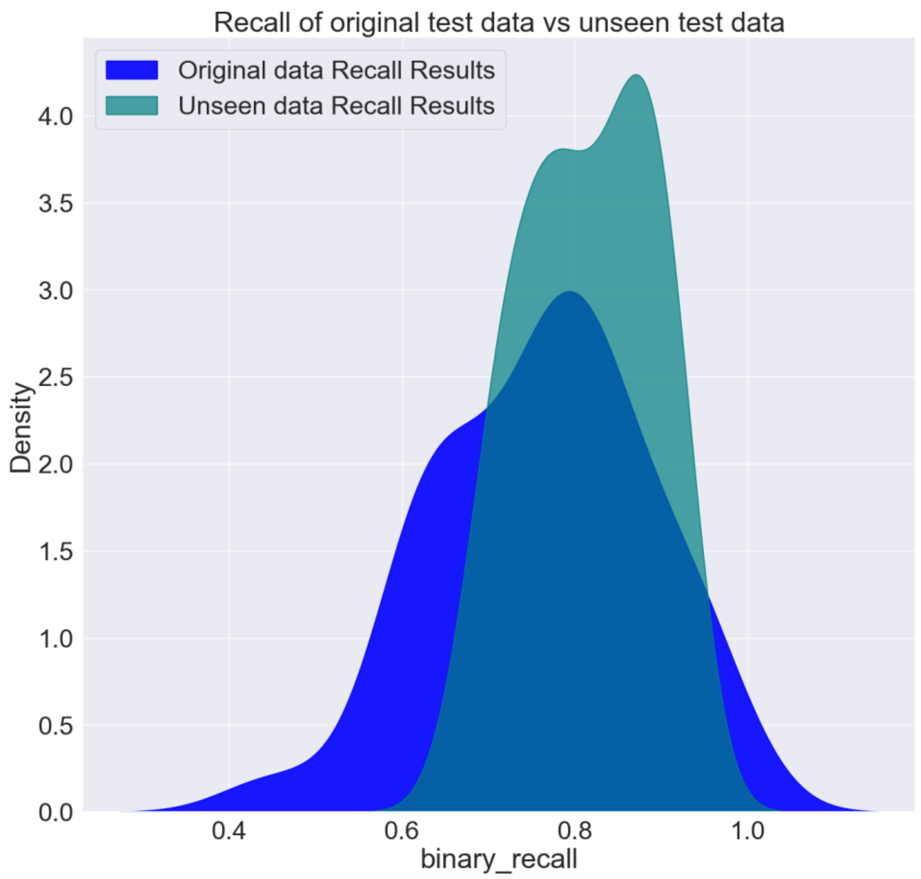
- TP = The number of predictions that are labelled 1 and classified as 1
- FN = The number of predictions that are labelled 1 and classified as 0



Looking at the binary recall between both models, the results are reflective of the trends seen in the confusion matrices and the plots representing them. The model results from the unseen data runs showed a higher proportion of TP and subsequently FP results. Ergo, the false negative results would have been lower. So the overall score for the model tested against unseen data appears to be better. Taking the means of the scores supports this. The Recall score for the model tested against the original data is 76%, while the result of the unseen test data is 81%. Additionally the spread of the unseen data results appears to be much more narrow. A dividing line could be drawn through the horizonal axis at y = 0.65 and there would be no values below this. While for the original test data the spread is much wider with the lowest result being around 50% (0.5)

Original test data	Unseen Test data
Proportion of positive results:	Proportion of positive results:
48.78%	60.84%

Original test data	Unseen Test data
Recall Score (%)	Recall Score (%)
76%	81%



A density plot of the recall results illustrates this. With the unseen result data having a much smaller spread. While the original test data is much more widely distributed. And has a lower density of results at any given score.
Unsurprisingly a t-test failed to reject the hypothesis that the original test data had higher binary recall over all.

Null Hypothesis:

$H_0 : \mu_1 \geq \mu_2$

Alternative Hypothesis:

$H_1 : \mu_1 > \mu_2$

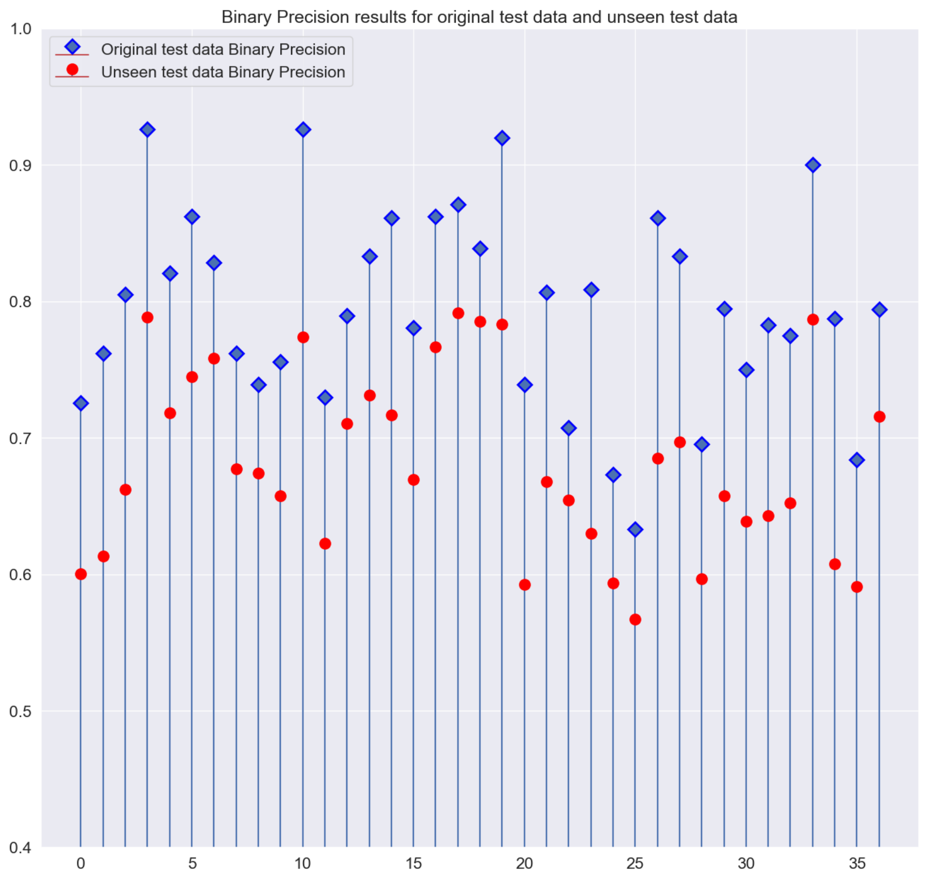
Resulting p-value at α 0.05, 0.9856649298.

Binary Precision data:

The calculation for Binary Precision is:

$$Precison = \frac{TP}{TP + FP}$$

- TP = The number of predictions that are labelled 1 and classified as 1 by the model.
- FP = The number of predictions that are labelled 0 and classified as 1 by the model.

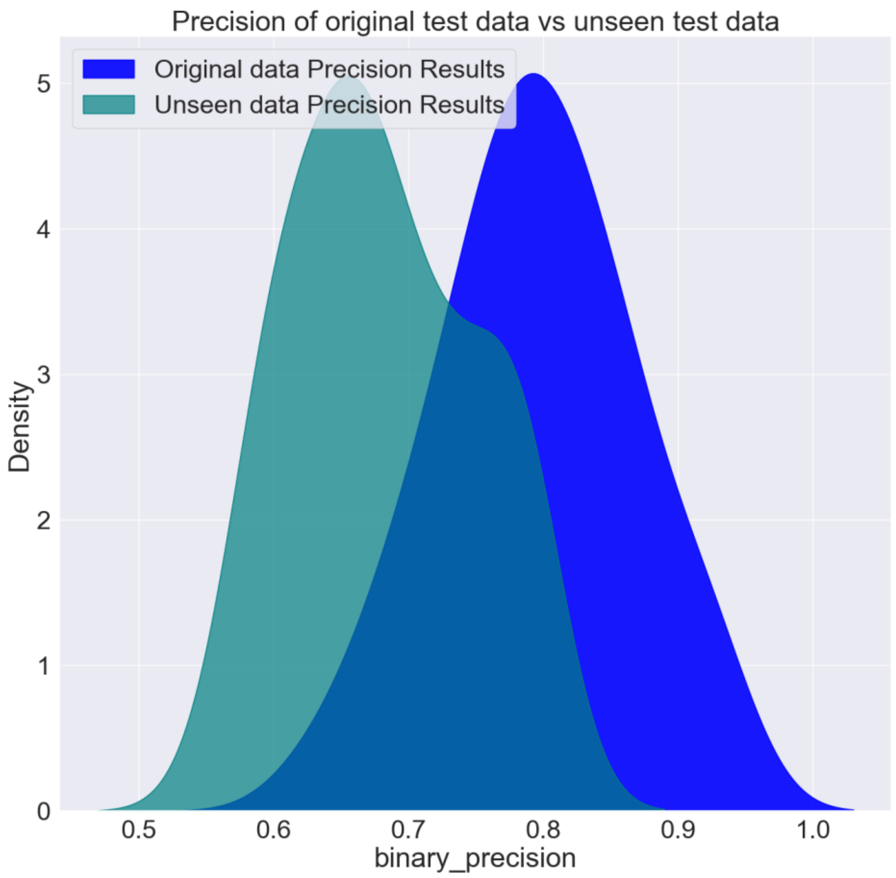


So how binary precision differs from recall is that it quantifies the total number of times the model labelled the result as positive. While binary recall is the sum of positive predictions and incorrectly classified positives(FN). In this case, the results of the model tested against unseen data appear to be worse than the original test results. The over all proportion of positive labels is higher in the unseen test data results :60.84% for the unseen test data versus 48.78% for the original test data.

The figure above indicates that the model performs much better in terms of recall when the original test data is used. With most of the data clustered around 80% (0.8) with and no results below 0.6 this idea is supported. While for the unseen test data most is clustered around 70%, (0.7). A table with calculated means supports this, the mean for the original test data was 79.52%, and for the unseen test data it was 68.17%.

Original test data	Unseen Test data
Proportion of positive results:	Proportion of positive results:
48.78%	60.84%

Original test data	Unseen Test data
Precision Score (%)	Precision Score (%)
79.52%	68.17%



A density plot for the precision data helps support the idea that the original test data yielded better precision results. The plot for the original test data in the above figure is shifted to the right, with the median at 0.8. And the left tail does not go further than the left tail for the unseen data and the right tail extends well past the right tail of the unseen data. The unseen data centers around 0.6, with a hump at 0.7, and a left tail ending at 0.5, with the right tail ending at 0.9. Despite the hump found in the unseen data, the overall shape of both plots is very similar with the most dense (the median for both) values having comparable proportions.

Unsurprisingly a t-test rejected the hypothesis that the original test data had higher precision over all.

Null Hypothesis:

$H_0 : \mu_1 \geq \mu_2$

Alternative Hypothesis:

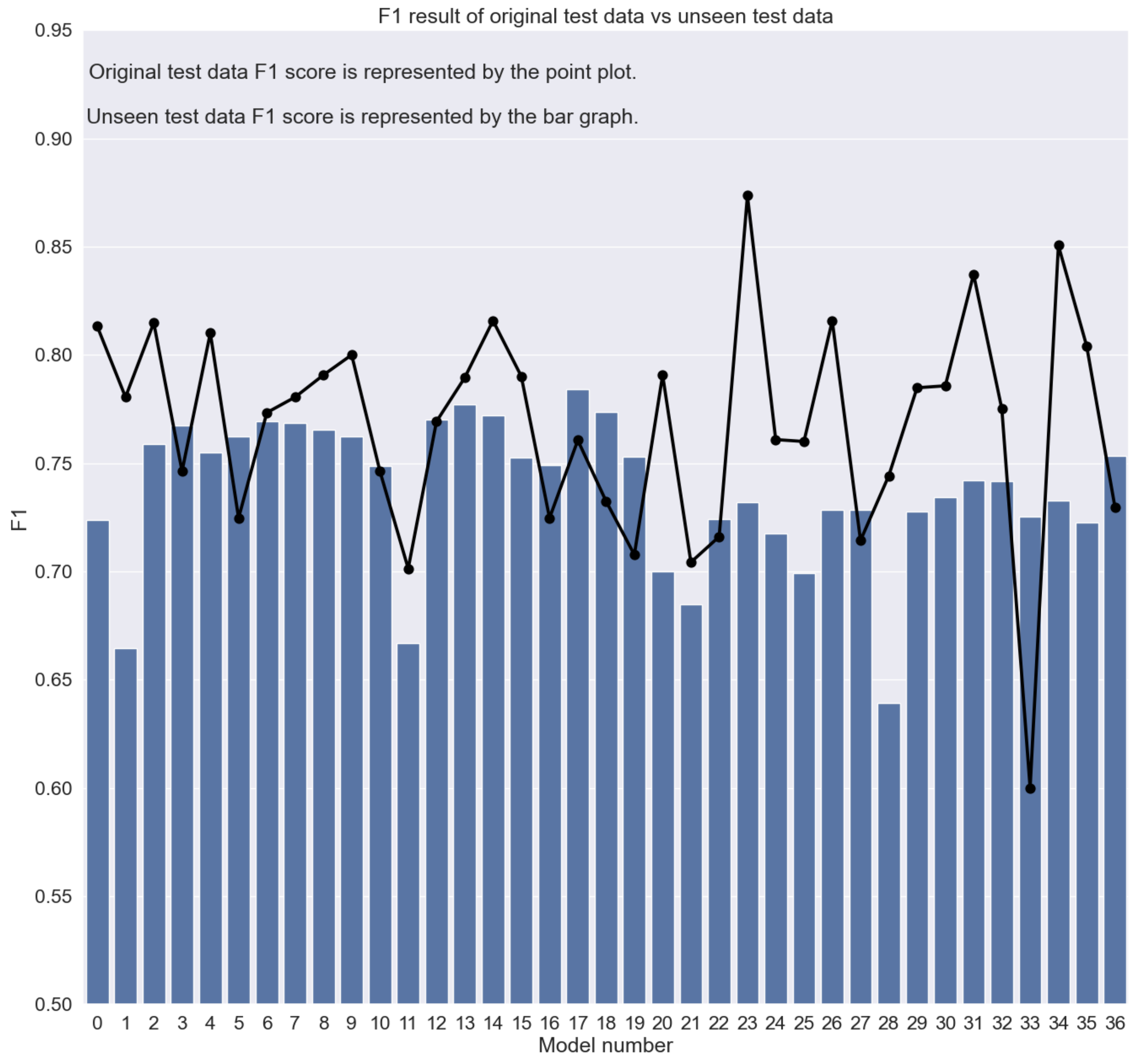
$H_1 : \mu_1 > \mu_2$

Resulting p-value at α 0.05, 0.00000005. The original model test data had better precision.

F1 scores data:

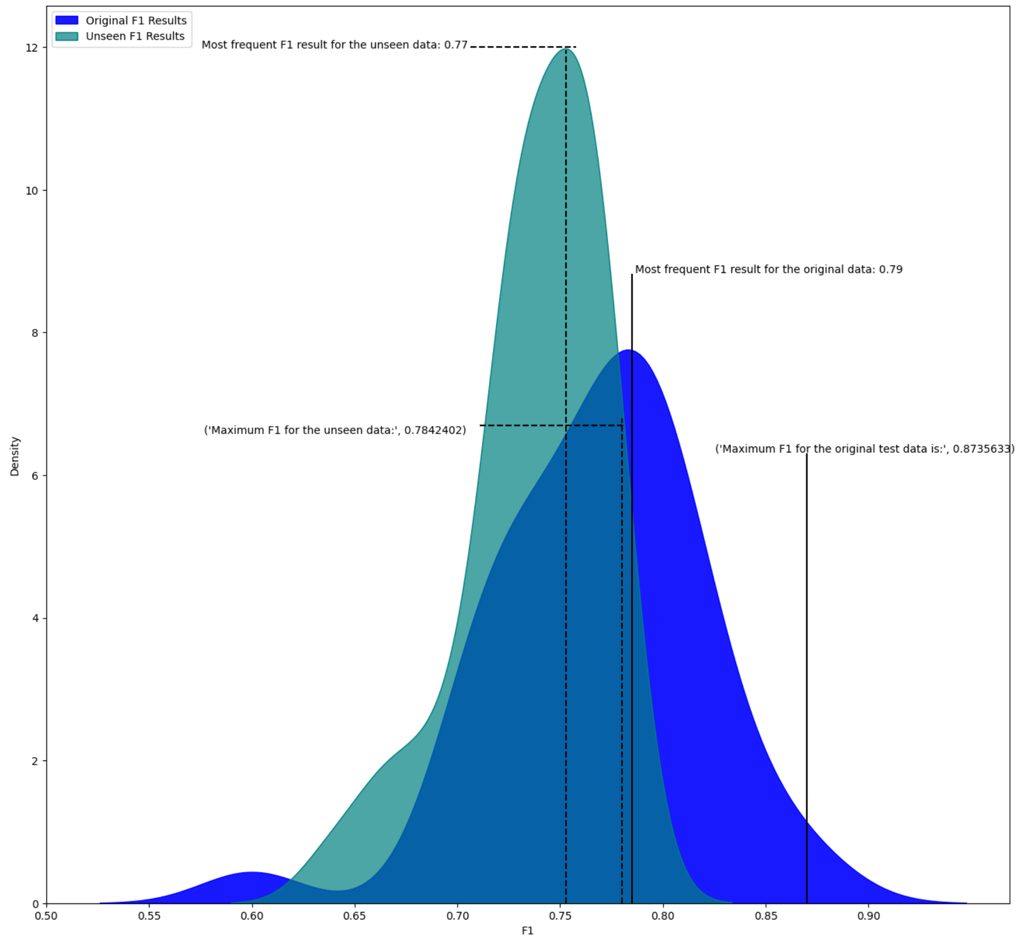
$$F1 = 2\,\frac{precision * recall}{(precision) + recall}$$

The F1score is the result of taking the previously calculated results and applying the above mathematical formula. It is a measure of overall model performance.



In general the above figure indicates that the results for the unseen data have a narrower spread than for the original test data. With the smallest value being around 65%, (0.65) and the largest being around 80% (0.8). While the original test data result has a minimum value at about 60% (0.6) and a maximum around 85% (0.85). This would suggest that the model performed better on the original test data than on the unseen test data. The table below shows the average score for original test data and unseen test data F1 results. The model performed had a higher average score for the original test data than for unseen. The original data had a score of 76.80% and the unseen test data had a score of 73.72%.

Original test data	Unseen Test data
Average F1 Score:	Average F1 Score:
76.80%	73.72%



The density plot illustrates the spread of the data well. While the original test data has a higher maximum, at 87.35%, the overall spread of the data is much wider with a minimum around 55%. And the density at which all results occur is also much lower. The test data does show a narrower spread, with a minimum around 60% and a maximum around 80%.

A t-test rejects the hypothesis that the original test data had the same F1 score as the unseen data.

Null Hypothesis:

$H_0 : \mu_1 \geq \mu_2$

Alternative Hypothesis:

$H_1 : \mu_1 > \mu_2$

Resulting p-value at α 0.05, 0.001465. So the original test data did have a higher F1 score.

For the overall results, the model performed better on data that came from the original data set that it trained on than on new completely unseen data. But from the many checkpoint models that were saved a ‘best’ model will be chosen and will be used for further testing.

Supplemental Section:

Note: Scikit-learn confusion matrix plots data in the following way.

$C_{0,0} = True\ Negative$

$C_{1,0} = False\ Negatives,$

$C_{1,1} = True\ Positive\ Values$

$C_{0,1} = False\ Negative$

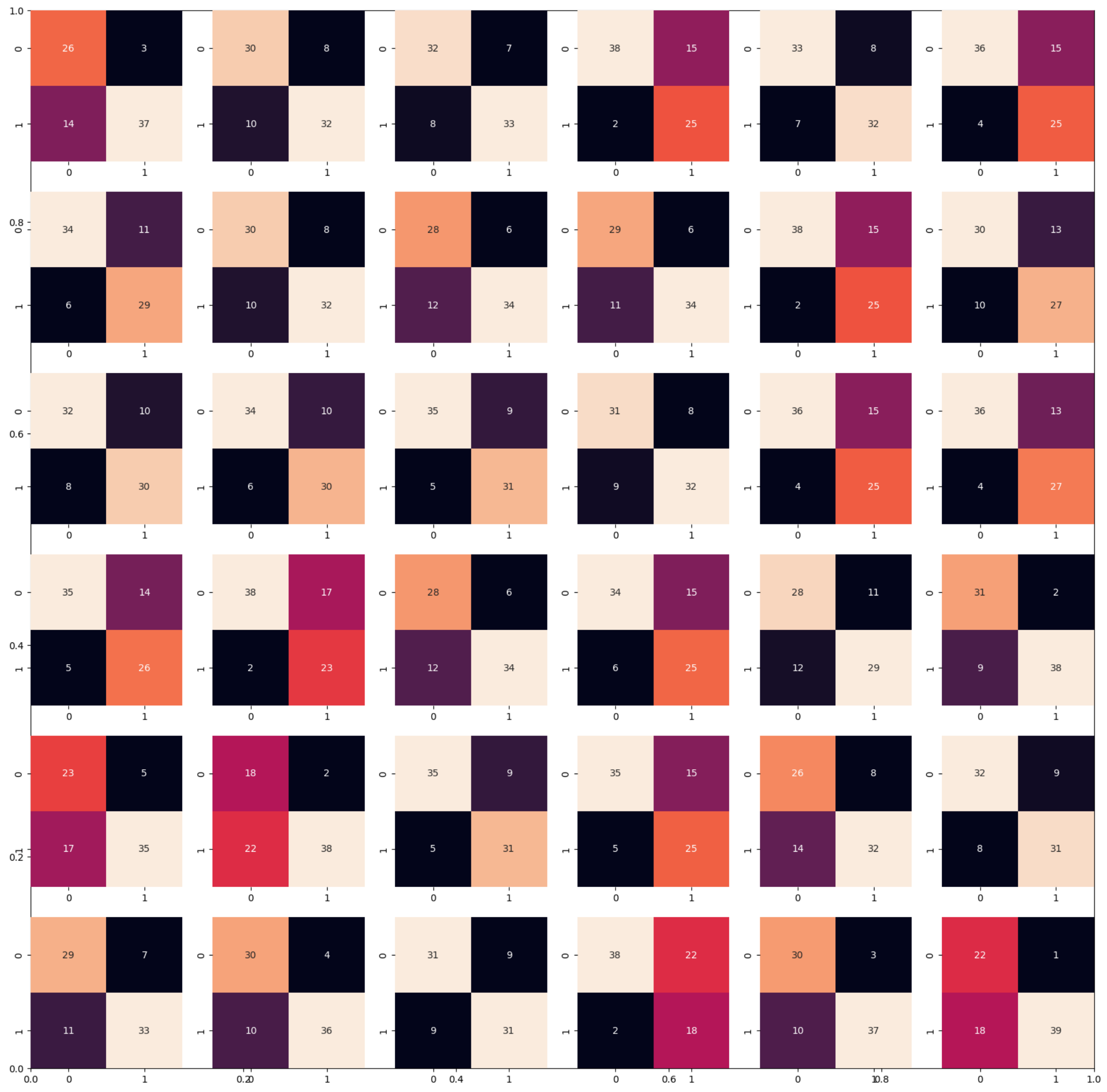


Figure 1 Confusion matrix for Original Data

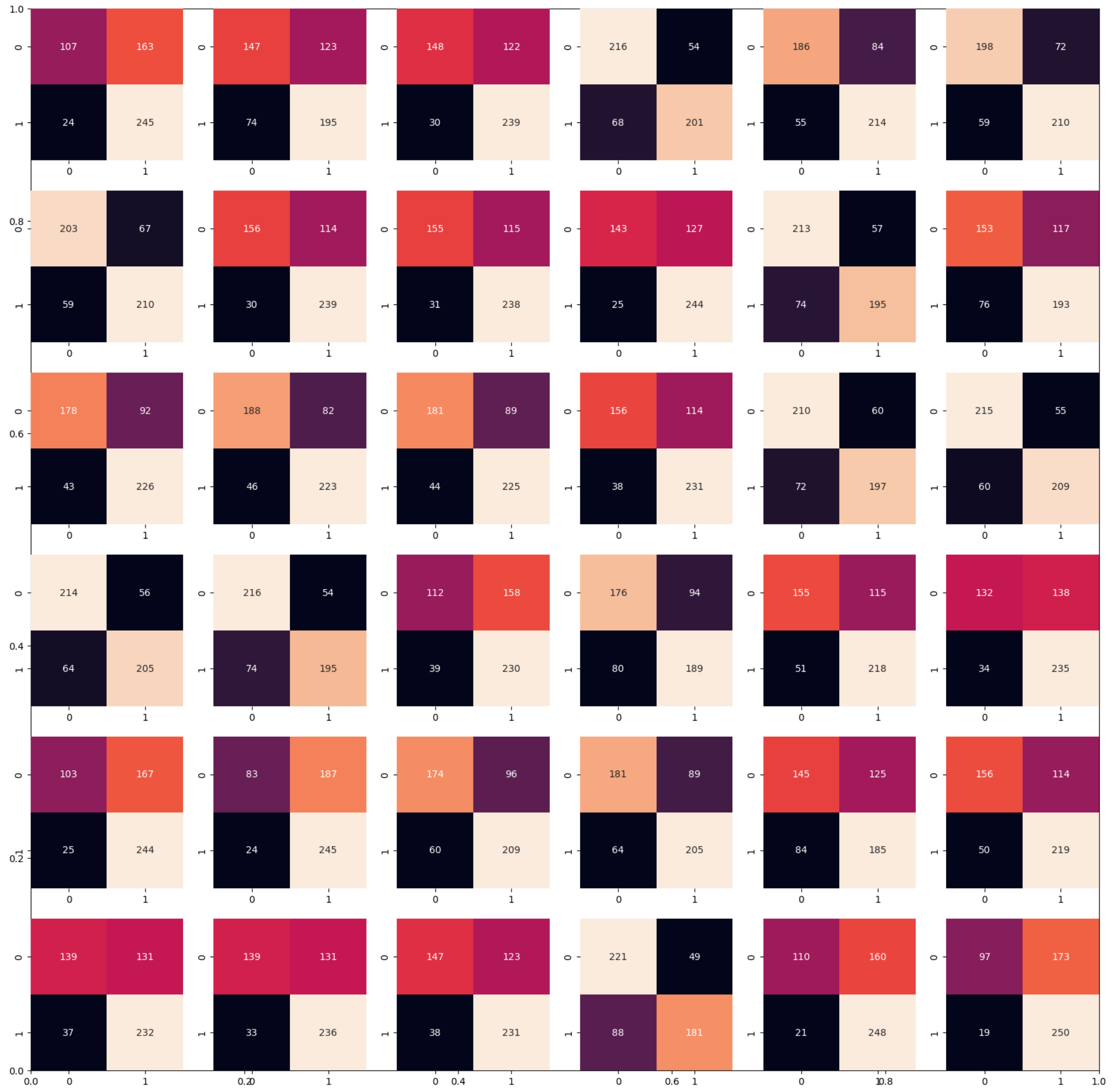


Figure 1 Confusion matrix for unseen data