Supervised Learning Final Project

For this project I wanted to tackle something meaningful to me. I followed the steps to locate existing datasets and found a set pertaining to spam emails. I am a huge fan of Kitboga and have experienced the effects that scams can have on individuals and their families. Using something small like this dataset I believed I could make a neural network using PyTorch to mitigate just how many scam/spam opportunities exist stemming from emails.

Overall Goal

The goal of this project is to minimize the opportunities for scammers to scam people via emails. Using data gathered regarding whether emails are spam or not and features that can be used to predict this, a neural network will be trained to detect whether incoming emails are spam or not spam (ham). Depending on the effectiveness of this neural network, this project will lead to less scammers getting a grasp on their victims while still letting emails through. Considering the authors of the dataset state the accuracy of their own model using this data, it can be used as a way to compare effectiveness.

The Dataset

https://archive.ics.uci.edu/dataset/94/spambase

I acquired my dataset from UCI ML Data Repository. I opted to use the "Spambase" dataset for my project. This dataset consists of continuous data with near zero cleaning required as their are no missing values. With 57 features and 4601 rows of data, the neural network will be able to optimally devise if any passed in email will be spam or ham. This data mostly consists of values denoting frequency of appearance when it comes to certain keywords or phrases along with capitalization. Using these features, we will then predict their validity. A flaw with this dataset is that the model will be trained for an individual's mailbox. There is a feature that utilizes information pertaining to the user it was based off "George" and "650" as the area code, but this can easily be switched out with a new user's information while using the same weights.

Importing the Data

To import the data I opted to just download the "csv" file and work from there. Included also are the documentation and descriptions of the data also provided from the source. This includes the sources, authors and past usages.

The actual data we need is located within `spambase.data`. This file is essentially a headless csv and therefore will be loaded in as such.

In [6]:
```python
# Loading in the data from the file using pandas
import pandas as pd

values = pd.read_csv('sup_learning/spambase.data',header=None)
values.head()
```

Out[6]:

	0	1	2	3	4	5	6	7	8	9	...	48	49	50	51	52	53	54	55	56	57
0	0.00	0.64	0.64	0.0	0.32	0.00	0.00	0.00	0.00	0.00	...	0.00	0.000	0.0	0.778	0.000	0.000	3.756	61	278	1
1	0.21	0.28	0.50	0.0	0.14	0.28	0.21	0.07	0.00	0.94	...	0.00	0.132	0.0	0.372	0.180	0.048	5.114	101	1028	1
2	0.06	0.00	0.71	0.0	1.23	0.19	0.19	0.12	0.64	0.25	...	0.01	0.143	0.0	0.276	0.184	0.010	9.821	485	2259	1
3	0.00	0.00	0.00	0.0	0.63	0.00	0.31	0.63	0.31	0.63	...	0.00	0.137	0.0	0.137	0.000	0.000	3.537	40	191	1
4	0.00	0.00	0.00	0.0	0.63	0.00	0.31	0.63	0.31	0.63	...	0.00	0.135	0.0	0.135	0.000	0.000	3.537	40	191	1

5 rows × 58 columns

Cleaning up the Data

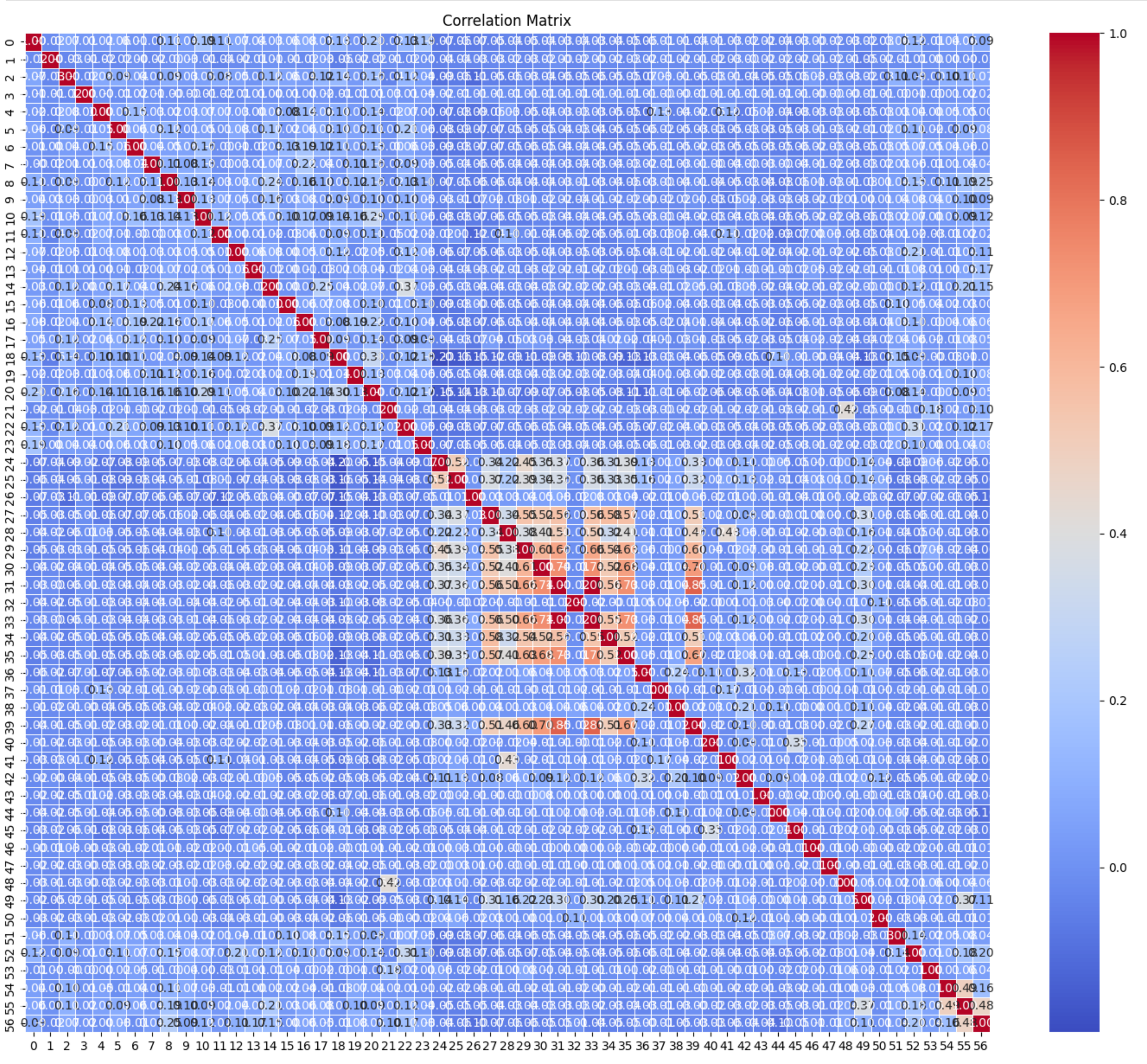
Now that we have the data loaded in, we should clean it, but this data is already pretty clean. There is no need to drop columns due to Null values, no need to drastically alter the data. However, for our dataset we want to still give our algorithm the best chance at having a successful prediction. To accomplish giving the best chance to our model we will use a method called `instance normalization`. We won't act on this until we are in our layers of the neural network, though. So until then, the data will stay in this relatively "raw" state.

In [8]:
```python
# we will need torch for this cell and onward
import torch
# I like building my pandas to tensor with numpy var types
import numpy as np

# we need to identify our targets column because they are not to be passed into the model
target_col = values.columns[-1]

# we want to shuffle our data because it is organized by spam or not
values = values.sample(frac=1,random_state=42)

# now we convert our DataFrame into a tensor that we can pass through to the model
X = torch.tensor(values.drop(target_col,axis=1).values.astype(np.float32))
y = torch.tensor(values[target_col].values.astype(np.int64))
```

Exploratory Data Analysis

We will want to fully know what we are getting into with this dataset and a great way to do that is through visualizations. One thing we can utilize with visualizations is seeing what features correlate with one another that may negatively impact the results.

In [51]:
```python
import seaborn as sns
import matplotlib.pyplot as plt

# drop the targets to only compare features
correlation_matrix = values.drop(target_col,axis=1).corr()
# we now plot the correlation matrix
plt.figure(figsize=(18, 15))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f", linewidths=0.5)
plt.title('Correlation Matrix')
plt.show()
```



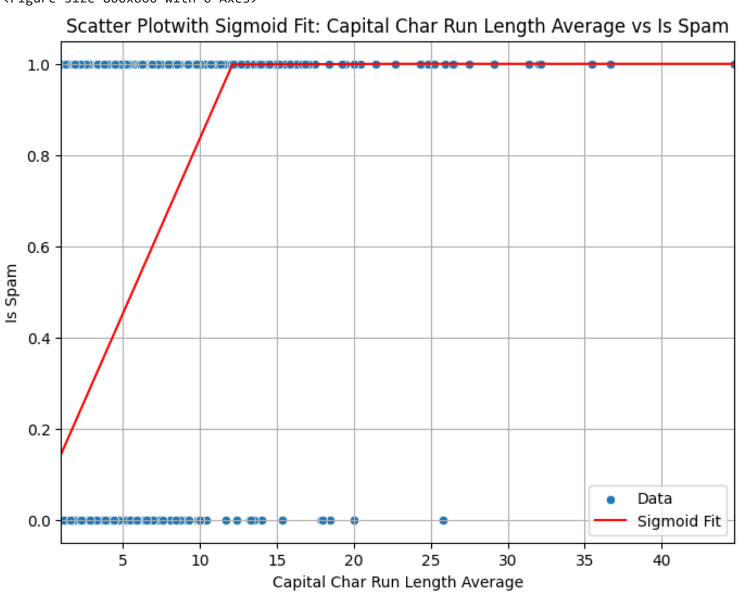
From our correlation matrix, you can see that most values do not bear much weight on others until you are in the middle region around column 30. This is because that is where a lot more of the personal information comes into play. Meaning that in these emails: if they mention your name, they most likely will also mention other personal information about you. Overall though you can see that these input features do not have too much of an impact on each other meaning that this data set will do well for our neural network. Using this graphic we could add our target column back into the mix to see features impacts but I think we can leave this up to the neural network to do on its own as all weights will be overwritten when it trains anyway.

Picking on Spammers

As we all know, a professional email should not contain all caps so lets see a scatter plot that highlights excessive use of caps in these emails and their correlation to if the email is spam or not.

In [62]:
```python
from scipy.optimize import curve_fit

plt.figure(figsize=(8, 6))

# assign the values for the coresponding columns
target_var = target_col
independent_var = values.columns[-4]

# Define the sigmoid function
def sigmoid(x, a, b):
    return 1 / (1 + np.exp(-a * (x - b)))

# Extract the independent and target variables from the DataFrame
x_data = values[independent_var]
y_data = values[target_var]

# Fit the sigmoid curve to the data
params, _ = curve_fit(sigmoid, x_data, y_data)

# Create a scatter plot using Seaborn
plt.figure(figsize=(8, 6))
sns.scatterplot(data=values, x=independent_var, y=target_var, label='Data')

# Plot the sigmoid curve
x_range = np.linspace(min(x_data), max(x_data), 100)
plt.plot(x_range, sigmoid(x_range, *params), color='red', label='Sigmoid Fit')

# set limits because some spam got out of hand
plt.xlim((np.percentile(x_data, 1), np.percentile(x_data, 99)))

plt.title(f'Scatter Plotwith Sigmoid Fit: Capital Char Run Length Average vs Is Spam')
plt.xlabel('Capital Char Run Length Average')
plt.ylabel('Is Spam')
plt.legend()
plt.grid(True)
plt.show()
```

```
<Figure size 800x600 with 0 Axes>
```



The sigmoid is extremely sharp due to how dense the Capital Char Run Lnegth Average is at the turning point. After cutting out outliers, you can see how much spammers use all caps in their emails, but it is not required in order to be spam. While this wasnt as drastically "all spammers use caps and real emails don't," this graph represents it would be it still shows just how much spammers love their capital letters of emphasis.

Preparing Our Data

While we do have our data set up in tensors and could hit the ground running already, we would be better off to split up our data into subsets for the varying stages of training. To do this, I am going to import the sklearn `train_test_split` function. This will make splitting things up much more effective. We will have 3 subset to work with now to maximize our potential training methods.

Split our raw tensors into training, validating and testing datasets

In [20]:
```python
from sklearn.model_selection import train_test_split
from torch.utils.data import TensorDataset,DataLoader

# create training/validation and test split from all data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.15, random_state=42)

# create the training and validation split from the previous training/validation set
X_train, X_valid, y_train, y_valid = train_test_split(X_train, y_train, test_size=0.2, random_state=42)

# Create training dataset and dataloader
train_tensor = TensorDataset(X_train, y_train)
train_loader = DataLoader(dataset=train_tensor, batch_size=10, shuffle=True)

# Create validation dataset and dataloader
valid_tensor = TensorDataset(X_valid, y_valid)
valid_loader = DataLoader(dataset=valid_tensor, batch_size=10, shuffle=False)
```

The Model

As stated previously, the desired model will utilize a neural network. Utilizing PyTorch as my desired library I will build out a custom neural network that uses a combination of linear, normalization, exponential linear unit (ELU), and softmax layers.

After experimenting with various layer widths, I had settled on a decently simple model containing 8 total layers. The final Softmax layer will be followed by an argmax function to enable backpropagation. When it comes to layer height, I prefer to condense my data rather than expand and so I stayed with using the natural dimensions of the data and worked it down until it could be categorized into spam or not with a boolean output.

In [9]:
```python
# now we can grab our feature size that we will use for our nn
feature_size = len(X[0])
```

Make our Neural Network

In [13]:
```python
# now we can actually build our nn
from torch import nn

class spam_detect(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_elu_stack = nn.Sequential(
            nn.Linear(feature_size, feature_size),
            nn.InstanceNorm1d(feature_size),  # Instance normalization layer
            nn.ELU(),
            nn.Linear(feature_size, feature_size // 2),
            nn.InstanceNorm1d(feature_size // 2),  # Instance normalization layer
            nn.ELU(),
            nn.Linear(feature_size // 2, 2),
            nn.Softmax(dim=1)
        )

    def forward(self,x):
        return self.linear_elu_stack(x)
spam = spam_detect()
# display our model to make sure it builds correctly
spam
```

Out[13]:
```
spam_detect(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear_elu_stack): Sequential(
    (0): Linear(in_features=57, out_features=57, bias=True)
    (1): InstanceNorm1d(57, eps=1e-05, momentum=0.1, affine=False, track_running_stats=False)
    (2): ELU(alpha=1.0)
    (3): Linear(in_features=57, out_features=28, bias=True)
    (4): InstanceNorm1d(28, eps=1e-05, momentum=0.1, affine=False, track_running_stats=False)
    (5): ELU(alpha=1.0)
    (6): Linear(in_features=28, out_features=2, bias=True)
    (7): Softmax(dim=1)
  )
)
```

Training the Model

Now that we have the model and the data, we need to actually put them together to get results. To do this we will create a training loop that trains using all of our training and validation data set created previously. We will also import a premade optimizer and loss function. You will notice that we load custom weights into the loss function and that is because we would rather have false negatives over false positives. One or two spam emails can slip through to make sure we do not block real or important incoming emails.

In [26]:
```python
# define our optimizer
optimizer = torch.optim.Adam(spam.parameters(),lr=1e-3)
# Define loss function with class weights
class_weights = torch.tensor([2, 1.0])
criterion = nn.CrossEntropyLoss(weight=class_weights)
```

Training loop

With the optimizer and loss function ready, we can now build out a training loop that will backpropagate so we can acquire more accurately tuned weights in our model. This process will also check to make sure that we do not over train by stopping early if the epochs have not been affecting the model much beyond a certain point.

In [28]:
```python
best_loss = float('inf')
patience_counter = 0
epochs = 150
patience = 5

for epoch in range(epochs):
    for batch in train_loader:
        X, y = batch
        yhat = spam(X)
        loss = criterion(yhat, y)

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

    # Calculate validation loss
    if valid_loader is not None:
        with torch.no_grad():
            valid_loss = 0
            for batch in valid_loader:
                X_val, y_val = batch
                yhat_val = spam(X_val)
                valid_loss += criterion(yhat_val, y_val).item()
            valid_loss /= len(valid_loader)

        # Early stopping based on validation loss
        if valid_loss < best_loss:
            best_loss = valid_loss
            patience_counter = 0
        else:
            patience_counter += 1

        if patience_counter >= patience:
            print(f'Early stopping: No improvement in validation loss for {patience} epochs.')
            break

    print(f'Epoch: {epoch} Training Loss: {loss.item()} Validation Loss: {valid_loss}')
```

```
Epoch: 0 Training Loss: 0.5995992422103882 Validation Loss: 0.4820128554784799
Epoch: 1 Training Loss: 0.33093395829200745 Validation Loss: 0.44786770773839346
Epoch: 2 Training Loss: 0.31561416387557983 Validation Loss: 0.4288427603395679
Epoch: 3 Training Loss: 0.31419044733047485 Validation Loss: 0.431896653733676
Epoch: 4 Training Loss: 0.40507462620735l7 Validation Loss: 0.41764804004113887
Epoch: 5 Training Loss: 0.3336279094219208 Validation Loss: 0.3910035185421569
Epoch: 6 Training Loss: 0.313774049282074 Validation Loss: 0.3880232425430153
Epoch: 7 Training Loss: 0.3135337233543396 Validation Loss: 0.388547342789324
Epoch: 8 Training Loss: 0.31964007019996643 Validation Loss: 0.4018107467814337
Epoch: 9 Training Loss: 0.3160257041454315 Validation Loss: 0.38716517407682877
Epoch: 10 Training Loss: 0.3135416507720947 Validation Loss: 0.37796604142913337
Epoch: 11 Training Loss: 0.45611175894737244 Validation Loss: 0.3962958673887615
Epoch: 12 Training Loss: 0.313565731048584 Validation Loss: 0.37626753841774374
Epoch: 13 Training Loss: 0.43803030252456665 Validation Loss: 0.3950536258613007
Epoch: 14 Training Loss: 0.31383782625198364 Validation Loss: 0.3727990686893463
Epoch: 15 Training Loss: 0.3328518867492676 Validation Loss: 0.3761472124842149
Epoch: 16 Training Loss: 0.46509501338005066 Validation Loss: 0.3980661517457117
Epoch: 17 Training Loss: 0.47615084052085876 Validation Loss: 0.3842629733719403
Epoch: 18 Training Loss: 0.31675419211387634 Validation Loss: 0.38470183218581766
Early stopping: No improvement in validation loss for 5 epochs.
```

Results

Now that we have the model trained up, we can use it. Let's pass in our test dataset we created earlier to see how well it performs.

In [29]:
```python
# we can take our softmax results and pass them through an argmax to get our boolean result for the emails
test_y_hat = spam(X_test).argmax(1)
```

Now that we ran our data through the model, we should see how it actually did.

In [35]:
```python
# now we can calculate how well our model has done
correct = (test_y_hat == y_test).float()

# calculate accuracy
accuracy = correct.sum()/len(correct)

# calculate false positives and false negatives
fp = ((test_y_hat == 1) & (y_test == 0)).sum().item()
fp_per = round(fp/len(y_test)*100,2)
fn = ((test_y_hat == 0) & (y_test == 1)).sum().item()
fn_per = round(fn/len(y_test)*100,2)

# calculate precision, recall, and F1-score
precision = (correct * test_y_hat).sum() / (test_y_hat.sum() + 1e-8)  # Adding small value to avoid division by zero
recall = (correct * y_test).sum() / (y_test.sum() + 1e-8)  # Adding small value to avoid division by zero
f1_score = 2 * (precision * recall) / (precision + recall + 1e-8)  # Adding small value to avoid division by zero

results = f"Accuracy: {round(accuracy.item()*100,2)}%, False Positives: {fp} ({fp_per}%), False Negatives: {fn} ({fn_per}%), F1 score: {f1_score.item()}"

print(results)
```

```
Accuracy: 94.36%, False Positives: 16 (2.32%), False Negatives: 23 (3.33%), F1 score: 0.9319372177124023
```

Consistency

Due to multiple variables, there is a chance that the results you see when rerunning these cells are different from what I got when writing up the report.

At the time of writing this report I got these results: `Accuracy: 94.36%, False Positives: 16 (2.32%), False Negatives: 23 (3.33%), F1 score: 0.9319372177124023`.

This is extremely good in my opinion but due to the nature of variability I decided to conduct an average of all stats. Here is what the average result looks like: `Accuracy: ~91%, False Positives: ~20 (2.9%), False Negatives: ~30 (4.3%), F1 score: ~0.90`

This means the model is relatively around what the original creators got with their models using this dataset. This was only accomplished after many attempts of fine tuning the neural network. I tried adding multiple layers, changing layer types, changing layer neuron count, altering training weights and tweaking batch sizes for the data loaders. This final result was fought for and relatively more consistent than my other trials.

Conclusion

This project accomplished a lot for me. This project managed to teach me new methods and further reinforce ideas I had previously about machine learning. A big piece I learned was to make my process a lot more clean and straightforward. Presenting my code in a fashion that could be properly followed instead of just diving in and jumping around like I used to do prior to this project. The organization managed to make this project not only more efficient and effective but allowed me to further the speed at which I can produce a model like this. On previous projects, I also never attempted to make such a powerful backpropagation training loop. I thought that the validation set was not entirely required and while that may be true, using it very much strengthens the model. One thing that I think could strengthen the model further is possibly using more cohesive layers. This model works but I think with more time the model could be fine tuned with not only layer height but also the actual layers inside of it. A lot of layers were used purely because I have used them before and mostly understand them and then I wanted to try something new with the normalization layers. The activation functions were already leaving my comfort zone so I did not want to stretch it further and risk over fitting. That would probably be the best bet for improvement, layer adjustments. I know tweaking the layers would require everything else to be tweaked as well but that is where I believe the weakest link is.